Dynamic Behavior Sequencing in a Hybrid Robot Architecture

THESIS

Jeffrey P. Duffy, Captain, USAF

AFIT/GCE/ENG/08-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCE/ENG/08-03

Dynamic Behavior Sequencing in a Hybrid Robot Architecture

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Masters of Science (Computer Engineering)

Jeffrey P. Duffy, B.S.E.

Captain, USAF

March 2008

AFIT/GCE/ENG/08-03

Dynamic Behavior Sequencing in a Hybrid Robot Architecture

Jeffrey P. Duffy, B.S.E.
Captain, USAF

Approved:

/signed/	03 Mar 2008
Gilbert L. Peterson, PhD (Chairman)	date
/signed/	03 Mar 2008
John F. Raquet, PhD (Member)	date
/signed/	03 Mar 2008
Maj Micheal Veth, PhD (Member)	date

Abstract

Hybrid robot control architectures separate plans, coordination, and actions into separate processing layers to provide deliberative and reactive functionality. This approach promotes more complex systems that perform well in goal-oriented and dynamic environments. In various architectures, the connections and contents of the functional layers are tightly coupled so system updates and changes require major changes throughout the system. This work proposes an abstract behavior representation, a dynamic behavior hierarchy generation algorithm, and an architecture design to reduce this major change incorporation process. The behavior representation provides an abstract interface for loose coupling of behavior planning and execution components. The hierarchy generation algorithm utilizes the interface allowing dynamic sequencing of behaviors based on behavior descriptions and system objectives without knowledge of the low-level implementation or the high-level goals the behaviors achieve. This is accomplished within the proposed architecture design, which is based on the Three Layer Architecture (TLA) paradigm. The design provides functional decomposition of system components with respect to levels of abstraction and temporal complexity. The layers and components within this architecture are independent of surrounding components and are coupled only by the linking mechanisms that the individual components and layers allow. The experiments in this thesis demonstrate that the: 1) behavior representation provides an interface for describing a behavior's functionality without restricting or dictating its actual implementation; 2) hierarchy generation algorithm utilizes the representation interface for accomplishing high-level tasks through dynamic behavior sequencing; 3) representation, control logic, and architecture design create a loose coupling, but defined link, between the planning and behavior execution layer of the hybrid architecture, which creates a system-of-systems implementation that requires minimal reprogramming for system modifications.

Acknowledgements

I would like to thank AFIT for the opportunity to pursue my Master's degree and CANIS for sponsoring this investigation. I would also like to thank Dr. Peterson for his guidance during the program. Most importantly, I thank my wife and daughter for their continued support and patience throughout.

Jeffrey P. Duffy

Table of Contents

List of Figures

List of Tables

Table Page

Behavior Representations

List of Symbols

Symbol		Page
D	List of Sensors	57
G	List of Abstract Goals	57
I	List of Initial Conditions	57
O	List of Output Conditions	57
C	List of System Controls	57
v	Vote Value	57
B	Behavior	61
g	Abstract Goal	64
L	Library of Viable Behaviors	65

List of Abbreviations

Abbreviation		Page
AP	Activation-Path	57
OP	Objectives Plan	64

Dynamic Behavior Sequencing in a Hybrid Robot Architecture

I. Introduction

Modern day mobile robots are becoming more advanced and capable of performing very complex tasks. These tasks range from the mundane to very dangerous and require that the robot react and make intelligent decisions about the environment and the intended task. To complete these tasks, the system requires reducing the human-in-the-loop control and a move towards a more reactive and goal-driven control. Hybrid robot control architectures separate plans, coordination, and actions into separate processing layers to provide deliberative and reactive functionalities. This approach promotes more complex systems that perform well in goal-oriented and dynamic environments.

In various hybrid architectures, the connections and contents of the functional layers are typically hardcoded, so changes within one layer require modifications in other layers. As systems grow and requirements change, the coupling of layers must not require major changes throughout the system to incorporate the change. This thesis proposes a layered architecture design using robust connections at each layer. By creating robust connections at each layer, the coupling between components and layers is less dependent, the software becomes more maintainable, and updates within layers cause minimal updating of others. The design of this proposed architecture is identified as the first of four Research Goals.

Research Goal 1: *Design an architecture that has distinct transitions between components (or layers) and can be implemented as a robust, modular software package that runs on different system configurations without major recoding.*

The majority of hybrid architectures link planning and execution layers through the use of task-level control languages [48]. These languages require that each behavior, or underlying functional characteristic, be expressed explicitly by the syntax of the language. This limits the implementation to the constructs of the language. This thesis also proposes an abstract representation that provides an interface for generically describing a behavior for sequencing (Research Goal 2), with the actual implementation left to the creativity of the behavior architect. Since the planning and execution of most hybrid architectures is controlled through the task-level control languages, the planning and execution layers are tightly coupled by the language. To loosen the coupling of the planning and execution layers, this thesis presents a dynamic behavior hierarchy generation algorithm that utilizes the behavior representation interface for translating high-level tasks through dynamically sequenced behaviors (Research Goal 3). By using the representation in a uniform manner, the planning and execution layers are loosely coupled by an abstract interface rather than an entire language. The behavior representation and hierarchy generation algorithm's implementation within the proposed architecture design enables a modular, robust system that requires reduced overall system maintenance during system modifications (Research Goal 4).

Research Goal 2: *Create an abstract behavior representation that provides an interface for describing a behavior's functionality without restricting or dictating the actual implementation or the use of the representation.*

Research Goal 3: *Describe a dynamic behavior hierarchy generation algorithm that utilizes the behavior representation interface for accomplishing high-level tasks by dynamically sequencing behaviors (i.e. dynamic hierarchy generation).*

Research Goal 4: *Combine Research Goals(1-3) to demonstrate a behavior representation that enables dynamic behavior sequencing within a robust and modular autonomous mobile robot architecture design, which requires minimal reprogramming for system modifications.*

All layered architecture creators describe the layers in detail [5, 11, 29, 46] and have shown that a layered architecture accomplishes the merging of reactive execution with deliberative planning. Although these accounts are detailed, many authors gloss over the actual mechanism used in transferring plans and state information between layers. For example, the establishment and selection of the connection between the sequencing and the controlling layers is never discussed by the creators. Even when the layer separations of the sequencing and controlling layers are clear, the mechanism for actual selection of behaviors for activation and deactivation and its translation to behavior execution is typically ignored. By all accounts, this appears to be the case because the connection is a mixture of trial and error or simply programmer hard-coding, and either approach is susceptible to programmer errors. By creating a representation for behaviors and linking the properties of the behavior functionality to the environment it expects and the tasks that it completes, this connection can be automated. Also, a control loop can be created that uses the representation in a uniform manner so that its functionality and the connection between sequencing and controlling will remain virtually unchanged during system modifications.

The remainder of this chapter provides an overview of the research presented in this thesis. First, a more detailed description of the problem to be solved is presented in Section 1.1. This is followed by the thesis objectives that aid in accomplishing Research Goals (1-4), and the general assumptions used.

1.1 Problem Statement

Autonomous robot systems are becoming more complex and more desirable for use in dynamic and unpredictable environments. These systems receive high-level taskings and attempt to complete the task using a combination of deliberative planning and reactive execution. However, most systems are designed for use in a specific environment and require major code restructuring when the environment or functional specifications change. Since these systems are designed explicitly for one purpose, the architectural design and implementation is very specific as well. The connections be-

tween different layers of a hybrid architecture are tightly coupled where small changes in one layer may result it major changes in another. This thesis proposes that using an abstract behavior representation to describe the behaviors increases the modularity of hybrid robot architectures and provides a means for a more robust functional decomposition of robot architecture design. This thesis also proposes a hierarchy generation algorithm that utilizes the abstract behavior description to dynamically select an arbitrated behavior hierarchy that accomplishes high-level goals and taskings without knowledge of the underlying behaviors' implementations. The architecture design, behavior representation, and hierarchy generation logic proposed in this thesis create a loose coupling, but defined link, between the planning layers and behavior execution layer of a hybrid architecture and its components. The accomplishment of the Objectives presented in the next section demonstrate the effectiveness of the proposed components at satisfying the Research Goals.

1.2 Thesis Objectives

The Research Goals presented above describe the high-level goals that guide the efforts of this thesis in solving the problem statement of Section 1.1. To satisfy the Research Goals, thesis Objectives are established that further guide this investigation. Since the Research Goals are high-level descriptions, each Objective aids in satisfying one or more goals. The following discusses each objective and its role in achieving the Research Goals.

Objective 1: *Show that the behavior representation and associated hierarchy generation algorithm can be applied in a hybrid robot architecture while adhering to the integrity of the three layer architecture (TLA) paradigm.*

Objective 2: *Show that there is a defined link between the Sequencer and Controller. This link must be an abstract mechanism that is robust and seamless.*

To satisfy Research Goal 1, a functional decomposition of a hybrid architecture is described with defined, robust coupling between components. Additionally, Objectives 1 and 2 also aid in accomplishing this goal by establishing a guideline for design and implementation. Since the case studies of Chapter V describe trial experiments for an implementation of the proposed architecture, behavior representation, and hierarchy generation logic, the successful results presented meet these Objectives and therefore satisfy Research Goal 4 as well.

Objective 3: *Show that an abstract behavior representation, which does not require the knowledge of low-level implementation details, can be applied as an interface to simple, complex and concurrent behaviors.*

Objective 3 aids in satisfying Research Goal 2 in that it does not rely on implementation details. Furthermore, by creating an abstract interface for behavior representation, other components can interact with the interface in a uniform manner without knowledge of its concrete implementation. This characteristic of the behavior representation leads to Objective 4.

Objective 4: *Show that a hierarchy generation algorithm can use the behavior representation to dynamically generate an arbitrated behavior hierarchy for accomplishing desired goals without a priori knowledge of system capabilities and behavior functionalities.*

Objective 4 aids in satisfying Research Goals 1 and 3 in that using the behavior representation without knowledge of the underlying implementation suggests that the hierarchy generation algorithm is not dependent upon behavior implementation. Additionally, the behavior representation acts as the defined mechanism for linking the planning of the behavior hierarchies to its actual execution. Objectives 3 and 4 are met together in the experiment described in Section 5.2, which:

1. Demonstrates that the behavior representation and hierarchy generation algorithm are independent of system capabilities and behavior implementation.

2. Demonstrates that the software implementation can be transferred from systems with the same behaviors but different capabilities and resource availabilities without code modifications.
3. Demonstrates that different system capabilities generate different behavior hierarchies.

Objective 5: *Show that a sequence of plans generates appropriate behavior hierarchies that are assigned at appropriate times to accomplish complex high-level tasking.*

Objective 6: *Show that dynamic system changes, such as hardware failures, or environment conditions generate new hierarchies if necessary.*

Objectives 5 and 6 aid in satisfying Research Goal 3 in that high-level tasks are accomplished through dynamic sequencing of behaviors in response to plans and unpredictable environments. These Objectives are met through the results of the experiments presented in Sections 5.3 and 5.4 respectively. These experiments address each Objective individually and show dynamic behavior sequencing through sequential plans and random hardware failures.

The successful completion of the Objectives set forth in this section satisfy the overall Research Goals for this investigation. To summarize, the Objectives that satisfy the Research Goals are: Objectives 1, 2, and 4 satisfy Research Goal 1; Objective 3 satisfies Research Goal 2; Objectives 4, 5, and 6 satisfy Research Goal 3; Objectives 1-6 collectively satisfy Research Goal 4. Therefore, case studies are developed to ensure that the Objectives are met.

1.3 Sponsor

This research is sponsored by the Cooperative Autonomous Navigation and Sensing (CANIS) lab task for the Air Force Office of Scientific Research (AFOSR). CANIS is located at the Precision Navigation and Time division of the Air Force Research Laboratories (AFRL/RYR) at Wright-Patterson Air Force Base. CANIS

requires autonomous navigation in dynamic environments to accomplish high-level tasks and goals for multiple agents. By creating the robust mechanism presented in this thesis for sequencing behavior execution within the agent architecture, the architecture can be applied to various systems with minimal change to the sequencing mechanism.

1.4 Assumptions

Although the techniques and methods presented in this thesis do not dictate the implementation language, we do assume that they are implemented using an object oriented design model. Since the research goals stress a robust and modular design, this paradigm is currently the accepted practice for accomplishing these goals. When possible, a generalized design concept is presented. However, when discussing experimental implementation, the C/C++ language is used. Therefore, a basic knowledge of the C/C++ language is assumed.

The design of the behavior representation and hierarchy generation logic anticipates that the Controller is implemented using the Unified Behavior Framework (UBF) described in [52] for its behavior execution. This framework promotes modularity within behaviors and is the foundation for this thesis's proposed behavior representation. Without the UBF, the seamless connection from the Sequencer to the Controller is lost. Other assumptions that are made about different components of the proposed architecture are discussed when necessary throughout the document.

1.5 Thesis Overview

This thesis is structured in the following format. This chapter introduces the problem and research goals at a high-level view. Chapter II provides an overview of behavior-based robotics and how they have evolved from strictly reactive control architectures to layered hybrid architectures that combine reactive control with deliberative planning and reasoning. Chapter III presents an overview of the various methods and techniques for representing and sequencing behaviors, as well as the link-

ing mechanism for coupling the layers of various architectures. The development of the behavior representation is presented in Chapter IV which also details the hierarchy generation logic that accomplishes the dynamic sequencing of behaviors utilizing the behavior representation. Three case study implementations are detailed in Chapter V, which demonstrate a proof of concept and validates the claims of dynamic sequencing, robustness and modularity. These case studies are presented as detailed experiments that include the purpose, implementation, results and a discussion of how the results support the claims of this thesis. Finally, Chapter VI concludes with closing remarks and possible future investigation for the complete implementation of the concepts addressed in this thesis and possible expansions to these concepts.

II. Behavior-Based Robotics Research

In autonomous mobile robot control architectures, the development has evolved from strictly reactive to behavior-based hybrid architectures. This chapter presents previous and current research in the area of reactive and behavior-based robot control architectures and their constructs for translating goals or desired functionality to robot motor settings (or behaviors). The focus is on architectures that implement a layered approach where a functional separation has been made between the sequencing element and the controlling element.

The background areas discussed in this chapter are divided into two sections. The first section discusses reactive robot control architectures and is followed by a discussion on hybrid/three-layer robot control architectures.

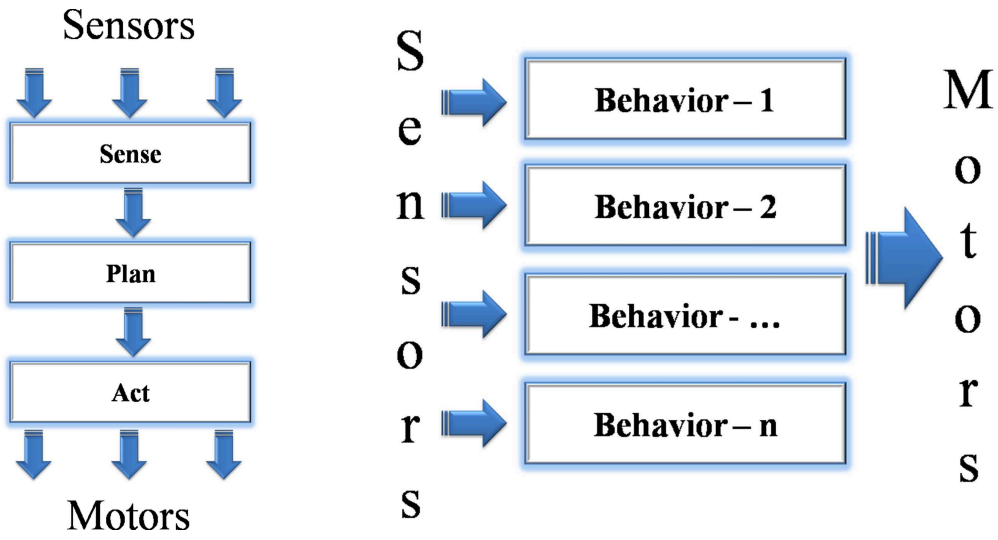
2.1 Reactive Control Architectures

Reactive control architectures emerged in response to the shortcomings of the Sense-Plan-Act (SPA) approach, prevalent in robot architectures previous to 1985 [17]. The SPA approach is a unidirectional and linear approach, Figure 2.1a. The linear control loop of SPA receives the sensor data (sense), computes a plan (plan), and sets motor settings accordingly (act). This paradigm requires an accurate world model and limits robot reaction times to the time required for updating the world model, computing an action plan, and appling the motor settings to react to the environment. The SPA approach does not perform well in dynamic and unpredictable environments, because the world model requirement forces strict environments that are known *a priori*. Additionally, the planning bottleneck from sensor input to computing plans limits the architecture to simple functionalities that require minimal environment or goal decomposition. Researchers identified that concurrent execution, rather than sequential execution, of the SPA paradigm components reduces the planning bottleneck within SPA.

To combat the shortcomings of sequential processing between sensing, planning, and applying motor settings, robot architectures began implementing concurrent pro-

gramming techniques. By programming the planning mechanisms in parallel and using different arbitration techniques, the computation bottleneck of maintaining a world model is reduced to the time required by the longest computational entity. With this new paradigm, planning mechanisms are decomposed into low-level entities that are identified as simple behaviors that accomplish specific functions. At the time, this type of decomposition was labeled with the contradictory term, *reactive planning* [17]. Reactive planning tightly couples sensing with action, which allows for the creation of behaviors that use sensor data to compute, in parallel with other behaviors, the motor settings appropriate for their functionality, Figure 2.1b. These specialized behaviors follow the idealized argument made by researchers, such as Braitenberg, that complex tasks are describable and performable by the combination of many simple tasks [7]. By eliminating the world model entirely and thus the internal state, there is more time for behavior computation, and each behavior uses only the sensor data needed for its computations. The actions are then based on the perceptions that are directly linked to the behavior and the actions become reflexive to the environment instead of trying to model it. Tasks are decomposed into a collection of low-level primitive behaviors that are typically stimulus-response pairs [3].

The first architecture to diverge from the SPA approach was the Subsumption architecture developed by Brooks [8]. Figure 2.1 shows the paradigm shift from the sequential flow of information processing from the sensors to the motor settings to a paradigm that uses concurrent task-achieving behaviors. In this architecture, each level (or behavior) independently processes the sensor data and suggests motor settings to achieve its intended functionality. Behaviors are an approximation process that works by dismantling world states into sub-states. The arbitration technique used is priority based where the higher levels, which are more complex behaviors, subsume the lower levels, or the lower levels, which respond to faster environment changes, inhibit the higher levels. Arbitration mechanisms are the main component for differences between the reactive control architectures that followed Subsumption.



(a) SPA (b) SPA Paradigm Shift

Figure 2.1: Paradigm shift from the a) sequential sense-plan-act paradigm to the b) concurrent, task-based action decomposition

By changing how active behaviors are selected and motor commands are set, a system gives the appearance of different functionality.

The arbitration techniques of reactive architectures consist of selecting the motor setting recommendations from behaviors based on highest priority [26], highest utility [40], and/or selecting combinations of motor setting [4]. Therefore, the decision as to which behavior's recommendation to execute is dependent upon the arbiter employed. Arbiters are typically characterized as competitive or cooperative. Competitive arbiters use a winner-takes-all technique to select one action recommendation. The Circuit architecture uses a hierarchical prioritization that is applied to different behavioral categorizations where each category contains either reactive behaviors or logical formalizations [26]. Action Selection allows the behaviors to enact an activation level when specific preconditions are met [31]. Of all the behaviors that have activation levels set, the behavior with the highest activation is selected. The Colony architecture differs from the Subsumption architecture by eliminating the ability of lower levels to inhibit higher levels [10]. Cooperative arbiters use action recommendations from multiple behaviors to generate one compiled action recommendation. The

Utility Fusion architecture evaluates the utility that results from taking a particular action and selects the motor settings from the actions with the highest utility that corresponds to each motor setting recommendation [40]. A Motor Schema architecture uses a vector summation of the action recommendations from each behavior [4]. The method used in selecting the active behavior dictates how the behaviors are preprogrammed and used.

Although the reactive architectures are shown to respond well in dynamic environments by using low-level task decompositions, they lack deliberation and planning abilities required to accomplish different or multiple long-term goals. These architectures were designed for specific tasks and environments and were bound by the architecture's inherent limitations. In [52], Woolley describes a software system called the Unified Behavior Framework (UBF) that allows seamless switching between arbiters and arbiter hierarchies. By establishing behavior architectures in the context of the UBF, one can dynamically interchange between architectures, capitalizing on the strengths of popular reactive-control architectures, such as Subsumption [8], Utility Fusion [41] and Colony architectures [9]. Thus, exploiting the tight coupling of sensors to actions that reactive-control architectures achieve. However, reactive controllers have the short-sighted mission of reacting to current environmental conditions and thus require deliberative and planning capabilities. In response to the *capability ceiling*, which is the lack of mechanisms for managing complexity, layered/hybrid architectures were developed to add multiple layers of planning and deliberation to the reactive control element of an architecture [17].

2.2 Layered Architectures

Layered architectures were created in response to the *capability ceiling* of reactive control architectures. Reactive architectures use a task-based decomposition to perform well in dynamic environments but lack the ability to deliberate and develop plans for goal accomplishment. Layered architectures utilize deliberative planning and reactive control together. Since deliberative planning and reactive control are equally

important for mobile robot navigation, when used appropriately, each complements the other and compensates for the others' deficiencies [41].

The following well-known architectures are presented as layered architectures. However, some are more of an analog to the layered paradigm and thus described accordingly. In all cases, each architecture is described and analyzed as it maps to the generic three-layer architecture presented by Gat [17]. This by no means covers all implemented layered architectures but provides an overview of the different motivations and approaches used in successful layered architecture implementations. Although the following presents an overview of the architectures, the main focus is on the connection between the "Sequencer" and "Controller" as plans are translated to behavior activation and deactivation.

2.2.1 Three-Layer Architectures. With the advent of hybrid robot control architectures that bridge reactive and deliberative functionality, many hybrid robot control systems have separated plans, coordination, and actions into separate processing layers. These layers can be generalized into three composite layers based on increasing levels of abstraction and temporal complexity: A reactive feedback control mechanism (Controller), a slow deliberative planner (Deliberator), and a sequencing mechanism that connects the first two components (Sequencer) [17]. The idea behind three-layer architectures(TLAs) is to merge deliberative planning and reasoning with a reactive control unit to accomplish complex, goal-directed tasks while quickly responding to dynamic environment changes. Figure 2.2 shows the layout of a TLA and how the layers interact with other layers. As shown, the architecture is hierarchical and its functional decomposition is based on temporal complexity. In [17], Gat introduces the TLA paradigm and describes how many researchers, up until 1998, collectively or independently migrated to this paradigm. Since then, layered architectures are still widely employed. Although not the only robot architecture employed, layered architectures have grown in complexity and have shown to be successful at producing robots with deliberative and reactive functionality.

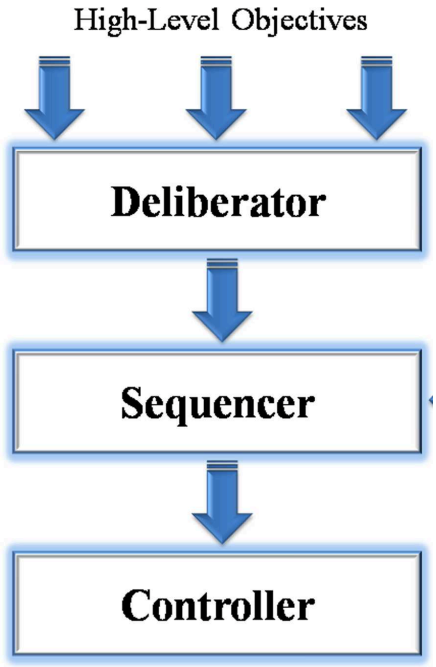


Figure 2.2: Three-Layer Architecture Hierarchy

Along with the three component layers, TLA focuses on minimizing the use of internal state. Minimizing the internal state reduces concurrency issues with the possibility that the environment represented in the internal state is not the current real-world situation. This may occur when the real-world situation changes before the internal state has completed its processing of the current state. Therefore the architecture relies on the functioning of the three layers/components. The following sections discuss the three layers: Controller, Deliberator, and Sequencer [17]. These layer descriptions are the basis for comparison and analogy for other layered architectures in the remainder of this chapter.

2.2.1.1 Controller. Layers within hybrid architectures are typically separated by abstraction and temporal complexity. The Controller is responsible for the low-level, reactive functionality that is accomplished in real-time without knowledge of high-level goals. The Controller is a reactive feedback control mechanism that performs the behavior-based functionality that is most commonly associated with the reactive control architectures discussed in Section 2.1. It maintains a library of primi-

tive behaviors that are the simplest decomposition of all the robot's functions. These primitives often ignore notions of internal state and instead focus on what is required in performing its intended behavior and most often rely on stateless sensor-based algorithms. The Controller is also tasked with anomaly detection and error handling. The primitive behaviors are typically execution threads that tightly couple sensor readings to actuators (i.e. motor settings). Each behavior reflects an appearance of a simple task, function or trait. Two examples of behaviors are `wall-following` and `obstacle-avoidance`. Behaviors are activated individually, or in combination, by the Sequencer or arbitration techniques producing simple or complex behaviors that perform desired tasks or accomplish specific goals.

The typical metric for identifying a behavior or function as a component in the Controller is temporal complexity. Since the function of the Controller is to utilize a library of sensor-to-actuator pairs, the computation time of the behavior cannot out live the actual state that it is reacting upon. Additionally, each behavior must allow bandwidth for other behaviors to react to the same environment. The likelihood of a dynamic state that changes before a behavior reacts is motivation for avoiding the use of an internal state. Behaviors are reactive to the current state and ideally avoid any internal state calculations. Other mechanisms within the layered architecture are charged with monitoring the state, namely the Sequencer and Deliberator.

2.2.1.2 Deliberator. The highest layer in a TLA is the Deliberator. The Deliberator is used for planning and predicting the future. This layer is the mechanism for deliberative computations that are time-consuming and consist of decomposing the high-level goals into discrete steps. In some architectures, the Deliberator may also answers queries made by the Sequencer. Goals and plans are decomposed by the Deliberator to the abstraction level of the Sequencer. Then, these decompositions are passed to the Sequencer for further decomposition. Since the Sequencer is at a lower level of abstraction and temporal complexity, the Sequencer accomplishes many iterations of behavior transitions before any results occur. This allows for the Delib-

erator to perform time-intensive computations through multiple threads of execution for future plans while the Sequencer executes and monitors current plans.

2.2.1.3 Sequencer. The sequencing layer (or Sequencer) transforms the plan output from the Deliberator to the actions of the Controller and maintains the state as the controller modifies it. The Sequencer also maintains the abstract represention of the state and when it notices events in the environment that conflict with the plans of the Deliberator, it sets a replanning trigger within the state for the Deliberator to handle. The Sequencer's job is to select the behaviors that the controller uses to accomplish the objectives set forth by the Deliberator [17]. This requires that the Sequencer set parameters for the behaviors and activates (and deactivates) the behaviors at strategic times to meet objectives. To do this, the Sequencer must monitor and update the state as appropriate. As seen in Figure 2.2, aside from sensors updating their data in the state, the Sequencer also sends information into the state. This allows for the setting of parameters and state variables that behaviors and other layers use. It responds to the current situation, or short-term past, and generates an appropriate behavior hierarchy to act upon the situation. The Sequencer has the ability to work on multiple, parallel interacting tasks and thus dispatches behaviors to accomplish each task in parallel. Although the Sequencer is afforded more computation time than the Controller, it performs computations relative to the rate of environmental change. The Sequencer selects behavior hierarchies that accomplish desired goals/objectives within the current environment. If the environment changes, the Sequencer is responsible for activating a different behavior hierarchy as appropriate.

This thesis focuses on the communication link between the Sequencer and the Controller. Therefore, the sequencing mechanism of other architectures, or the behavior activation/deactivation techniques are discussed more thoroughly in the following sections.

2.2.2 Saphira. The Saphira architecture integrates three concepts required for robot autonomy: The ability to attend to another agent, to take advice about the environment, and to carry out assigned tasks. To succeed in these areas, Saphira uses the concepts of coordination of behavior, coherence of modeling, and communication with other agents [29].

Saphira achieves coordination by coordinates its activities through a layered abstraction that helps to make the complexities of activities more manageable. However, just the control level, which is the layer that achieves coordination, is explicitly defined in [29]. At the control level, Saphira uses a behavior-base methodology. Each behavior creates a desirability function and is combined using fuzzy logic and "defuzzified" to create an action for execution. The architecture makes use of two respresentations of space, these are the Local Perceptual Space (LPS) and the Global Map Space (GMS) [27]. The LPS is an egocentric coordinate system that is useful for controlling purposeful movement by keeping short-term track of the robot's motion, fusing sensor readings, and registering obstacles to be avoided. The GMS represents the global environment by keeping track of *artifacts*, objects within the environment to include among others doors, rooms, and corridors. These *artifacts* are used in determining the action to take, thus combining strategic goals and prior knowledge that the Controller utilizes. Although not explicitly identified in the Saphira architecture, a sequencing layer could use these artifacts to dispatch appropriate behaviors according to real world and internally represented environments. The sequencing layer is loosely represented by the Procedural Reasoning System (PRS) (see Section 3.1.4). Additionally, these *artifacts* are maintained by the planning/Deliberator layer and are used for planning complex goals. If the authors in [29] separated these actual layers, then the need for the behavior itself to have a "context of applicability" is eliminated. However, Saphira's implementation of simple behaviors and fuzzy logic enables context dependent blending of simple behaviors to create complex composite behaviors. Figure 2.3 shows the Saphira architecture and highlights how the architecture fits the three-layer architecture paradigm.

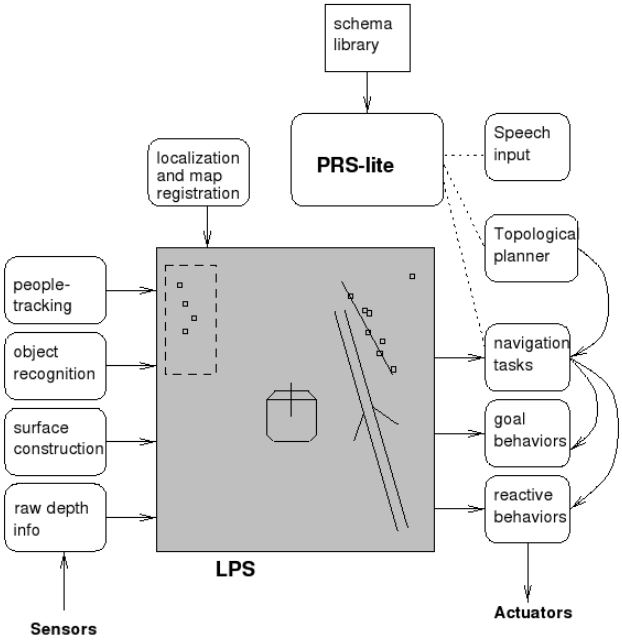


Figure 2.3: Saphira system architecture. Perceptual routines are on the left, action routines on the right. The vertical dimension gives an indication of the cognitive level of processing, with high-level behaviors and perceptual routines at the top. Control is coordinated by PRS-Lite, which instantiates routines for navigation, planning, execution monitoring, and perceptual coordination [28].

Saphira's Controller layer contains the Advanced Robotics Interface for Applications (ARIA) software package that controls lower-level routines [27]. These routines range from the hardware specifics of setting motor settings to the selection of behaviors. ARIA uses a *resolver* class to act as an arbiter for selecting the action recommendations of the active behaviors. Although the user can create custom arbiters, just one *resolver* can be used on the robot during execution [33]. The behaviors within the control layer of the Saphira architecture are behavior-based [29]. These behaviors are either reactive or goal-seeking behaviors that use inputs from the *state reflector* to accomplish the intended task. To build more complex behaviors, the Saphira architecture uses *context dependent blending* of simpler behaviors. The Controller uses the arbitration of the current activated behaviors to affect the motor settings. Therefore, a *controlling executive* (or Sequencer) is responsible for activating and deactivating behaviors.

PRS-Lite provides the sequencing component for the Saphira architecture [29]. PRS-Lite is Saphira's *controlling executive* that responds to the changing environment by instantiating routines for navigation, planning, execution monitoring, and perceptual coordination. While remaining responsive to the changing environment, the PRS-Lite attempts to accomplish event-driven and goal-driven activities. Saphira's sequencing layer also monitors behavior performance and activates and deactivates the behaviors when they are accomplished. Monitoring allows for sequential activation of behaviors to accomplish tasks or to identify failures. For instance, if the goal is to enter a room and pick up a cup of coffee. The Sequencer would activate a `go-through-door` behavior before activating the `pick-up-cup` behavior since the robot must go into the room before it can pick up the coffee. However, if the door is closed, then the Sequencer must identify that the `go-through-door` behavior has failed and update the state to indicate a closed door. This functionality requires that the Sequencer "know" everything about each behavior. Whenever a new behavior is added, the Sequencer must be modified. This is undesirable since the ideal sequencing mechanism is a modular programming entity.

The closest analog to a Deliberator is Saphira's mechanism for communicating with other agents and decomposing the input to attainable goals. Saphira uses speech recognition and text-to-speech to communicate with other agents [29]. The problem of communication is decomposed into the types of commands that the robot can receive: direct motion, sensor-based movement, task, and information. The direct motion commands require basic mechanical features such as move forward, turn around and stop. A sensor-based command requires sensors to accomplish the task such as following, avoiding, etc. Task commands require setting goals to be accomplished, such as "go get ...." Finally, information commands are the most abstract since they require the robot-centered representation of the world to coincide with the human-centered representation. If the information command was "this is a car", the robot and human representation of a car must be the same to make any sense.

2.2.3 Three Tier Robot Control (3T). The 3T architecture's name is derived from its separation of the general robot intelligence problem into three interacting layers or tiers (3T). This architecture was designed from the outset to control physical agents and mobile robots [5]. 3T combines deliberation and reactivity to enable robots to perform tasks in unpredictable environments. The three tiers, or layers, of the architecture are: Deliberation, Sequencing and Reactive Skills (Figure 2.4).

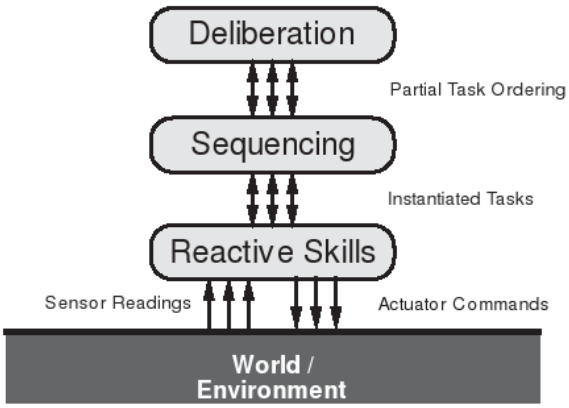


Figure 2.4: The 3T Intelligent Control Architecture [5]

The first, or lowest of the three tiers is a reactive skills tier that is coordinated by a skill manager (Controller). This skill manager is used as an interface between the skills (behaviors) and the rest of the architecture. It acts as a wrapper for implementing specific skills into the architecture. It also forces a standard interface to ensure a seamless connection within the rest of the architecture. Although this forced standard is convenient for this architecture's development, it requires the inputs and outputs of all skills. The skill manager then routes these outputs to the inputs of other skills. This dependence can cause a domino effect of problems within the skills if a skill's outputs are modified, therefore, not making each skill entirely independent.

The second tier within the 3T architecture is the sequencing tier (Sequencer). This tier activates and deactivates skills to accomplish specific tasks and to react to the environment with the use of Reactive Action Packages (RAPs). Each RAP is a description of how to react to the environment or accomplish a task(s). This requires the Sequencer to have a library of RAPs that are appropriate for specific situations and tasks that the robot must routinely perform. The Sequencer (or RAP interpreter) decomposes RAPs into other RAPs and activates a specific set of skills in the Controller based on the RAP decomposition and the intended tasks to be accomplished. RAPs are discussed more thoroughly in Section 3.1.1. The Sequencer also monitors the actions and environment for specific world conditions and events that require new behavior activations. In [5], Bonasso never mentions how the Sequencer reacts if it cannot find a RAP that can handle a given situation. This may occur as a result of the initial intended environment changing or the developer overlooking the specific situation. Since the RAP interpreter is simply a hard-coded description of how to accomplish common tasks, any changes to the environment or intended purpose of the robot requires code modification in the Sequencer.

The deliberation layer (Deliberator) is the third tier and the highest level of abstraction in the 3T architecture. The Deliberator handles the planning capability. It reasons in-depth about goals, resources and timing constraints. This reasoning is accomplished through the use of an adversarial-planner. The adversarial-planner allows

for the option of multi-agent coordination. 3T also has support for three types of planner control: top-down, problem-solving, and concurrent planning and execution [6]. This support adds for the ability of robust sequencing between planning methods for the accomplishment of different goals/tasks. However, there is some ambiguity in the Deliberator's error checking responsibilities. Since the Deliberator operates at the highest level of abstraction, it may not anticipate a low-level error condition of possible future states or the decomposition of an employed RAP. When the Sequencer decomposes the RAP issued by the Deliberator, it should notify the Deliberator of specific goal altering events instead of relying on the Deliberator checking the state.

2.2.4 Task Control Architecture (TCA). The architecture known as the Task Control Architecture (TCA) was specifically developed for achieving multiple, complex tasks in a rich environment [45]. Simmons, et al. [46] discuss how TCA is used with the development of an office delivery robot named Xavier. TCA consists of four independent abstraction layers (Figure 2.5). The highest layer has the most abstraction, looking at the big picture, and the lowest layer has the least abstract dealing with low-level details. These four layers, from highest abstraction to lowest abstraction, are task planning, path planning, navigation and obstacle avoidance. Layer independence allows each layer to be implemented without the other layers, which enables the layers to perform independently and reject any "advice" from higher levels if that advice inhibits the desired functionality of the layer. For example, if the path planner advises a path that has an obstacle, the obstacle avoidance layer can reject the "advice" and veer off course to avoid the obstacle. Although Simmons suggests and describes just four layers, he ignores the fact that the very basic behavior-level controls of the servomotors could be considered another layer. However, for the TCA architecture, the servo controls were supplied commercially. Additionally, the claim of independent layers is used loosely since there is bidirectional communication between the layers. When signals are propagated up the hierarchy (e.g. avoid obstacles trapped in local minima), this cross communication suggests dependence rather than independence.

Task Planning (PRODIGY)
Path Planning (Decision-Theoretic Planning)
Navigation (POMDP)
Obstacle Avoidance (CVM)

Figure 2.5: The Task Control Architecture (TCA) [47]

In regards to error handling/avoidance, the high layers aid the lower layers by avoiding situations that potentially result in difficulty for lower layers to handle. This may help with initial avoidance of errors, but leaving the error handling/avoidance at such a high level of abstraction may cause unsolvable failures at the lower levels. Another form of error handling could be seen in the heavy use of models and internal representation. This can help within the higher (deliberative) layers to correct for noisy or incorrect sensor readings. However, too much reliance on models and internal representation may move the robot further away from a reactive behavior and more towards the task-plan-act robots, which may reduce efficiency.

Although this architecture has four control layers that interact with a web interface and commercial servo control software, some claim that there are essentially no layers in regards to a three-layer architecture [5]. To the contrary, by separating out different functionalities of the control layers, the architecture is analogous to the three-layer architecture paradigm. The deliberative layer contains TCAs opportunistic scheduling and probability portion of the path planning. The sequencing layer contains the mechanism for monitoring the progress of the robot and making action adjustments for errors in task completion (e.g. failed to meet a waypoint) and, also contains the functionality of the navigation layer. The Controller layer contains behaviors based on the obstacle avoidance techniques and any motion activation that performs basic movements. The main drawback of this architecture is the lack of a programming model for adding system functionality and the lack of an explicit representation for expressing relationships among tasks [5]. Therefore, all changes

within the system requires the programmer to mentally compile and implement the appropriate control structures into the correct function calls.

2.2.5 SSS. The architecture discussed in this section is an acronym for "servo, subsumption, symbolic" system (SSS) [11]. SSS is a subsumption-based, three-layer architecture that capitalizes on the features of servo-systems and signal processing with multi-agent reactive Controllers and state-based symbolic AI systems. The layer hierarchy is established through quantizing space then time. At the lowest level (servo), the domain of space and time are continuous where the domain of space and time are discrete at the highest level (symbolic). The symbolic layer is responsible for strategic navigation where it maintains a map and a mechanism for activating and deactivating the subsumption layer modules (behaviors) after calculating the appropriate path. The symbolic layer also monitors for errors such as a *path-blocked* event where the robot has not reached its goal and has not made any forward progress. The subsumption layer contains a library of behaviors that it arbitrates just as the Subsumption architecture described in Section 2.1 based on the parameters set by the symbolic layer. Additionally, the servo layer contains behaviors that the subsumption layer activates and deactivates. The main functionality of the servo layer is to translate the desired translation and rotation speeds to actual motor settings.

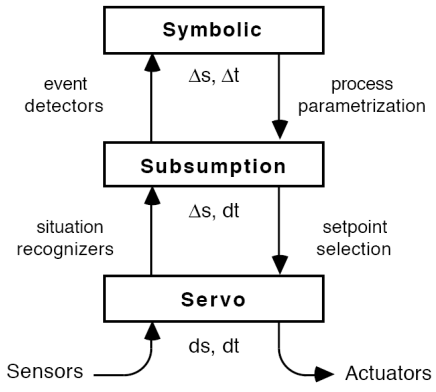


Figure 2.6: The SSS architecture combines three control techniques which can be characterized by their treatment of times and space. Special interfaces allow the layers of this system to cooperate effectively. [11]

Although the SSS architecture appears to have three layers, when applying the structure of TLA discused in Section 2.2.1, it has just two analogous layers. For the SSS architecture, the deliberative layer is only in charge of map generation and setting waypoints to navigate the terrain. The Sequencer monitors progress and sets the parameters for a Subsumption architecture. Since the subsumption arbitration and behavior selection is actually in a Controller's functional domain, the mapping can be moved to the sequencing layer, eliminating the need for an explicit deliberative layer. By organizing the layers according to computational time complexity, this architecture can benefit from current processor speeds and therefore capable of implementing a more sophisticated deliberative layer. Additionally, SSS has only been demonstrated on tasks involving pure navigation and its response to contingencies is limited by Subsumption's finite state machine model [5].

2.2.6 Overlooked Layer Connections. All layered architecture creators describe the layers in detail. However, many authors gloss over the actual mechanism used in passing from one layer to another. For example, none of the authors explicitly state how the connections are established and selected for linking the sequencing layer to the Controller. Even if the sequencing and controlling layer separation are not clear, the mechanism for actual selection of behaviors for activation and deactivation is typically ignored. By all accounts, this appears to be the case, because the connection is a mixture of trial and error or simply programmer hard-coding, which are both susceptible to programmer errors. By creating a representation for behaviors and linking the properties of the behavior functionality to state and goals, we can automate this connection and create a dynamic behavior hierarchy generation loop that remains unchanged during system modifications.

2.3 Summary

The *capability ceiling* of reactive controllers identifies the need for deliberative planning in conjunction with reactive control. This is accomplished through layered

architectures that have proven effective in solving problems and performing complex tasks. Each layered architecture uses some form of arbitration of behaviors that can be implemented uses the UBF described in [52], which is an ideal defining line of the Controller. Although not all of the layered architectures explicitly followed Gat's description of a TLA in [17], we have shown that they still have the elements to be categorized as a TLA. The difficulty in describing the architectures using the TLA paradigm is identifying the actual connection between the layers and how these layers communicate. Typically, the connections are hard-coded constructs like RAPs or trial and error programming until the system functions as expected. Specifically, the connection between the Sequencer and Controller layers usually required recoding whenever system modifications are made. Making modifications within the layered hierarchy in response to simple behavior changes requires extensive testing of modules that should not require changes as a result of changes outside of its focus. By creating a representation of lower-layer functionalities (or behaviors), we can create a dynamic behavior hierarchy generation loop that is not effected by addition, deletion or modification of behaviors within the Controller layer. Chapter III presents the current research in autonomous agent planning with a focus on planning algorithms used to select behaviors for activation and deactivation. Included in the chapter are the current constructs used for behavior representation.

III. Agent Behavior Planning Background

Component integration in hybrid architectures translate high-level planning to low-level executing using various techniques. However, as identified in Section 2.2.6, descriptions of Three Layer Architectures do not explicitly address the connections between these components (namely the Sequencer and Controller). Additionally, the connections are sometimes interleaved and the division of control is unclear. Our mechanism for formalizing these connections creates a distinct division between layers. This facilitates in the development of a robust and modular hybrid architecture. The mechanisms and techniques described in this chapter address various concerns and considerations in regards to the translation from planning to execution to goal completion.

This chapter presents previous and current research in the area of translating goals or desired functionality to robot motor settings (or behaviors). In other words, coupling the sequencing layer (Sequencer) to the Controller using different planning techniques and behavior representations. The following sections present the techniques used in various systems to interface deliberation and control, as-well-as the different methods for describing or representing actions/behaviors.

3.1 Plan Execution Languages

Many autonomous robot systems employ sophisticated plan execution systems that continuously interleave planning and execution [20], also referred to as task-level control languages [48]. The following plan execution systems typically support concurrent execution, decomposition of tasks into sequences of subtasks, resource management, execution monitoring, failure detection and analysis, exception handling, and execution recovery strategies [20]. The majority of hybrid architectures link the Sequencer and Controller through the use of these languages. These languages require that each behavior (e.g. task-net [16]) is expressed explicitly by the syntax of the language. Although the behaviors are limited to the constructs of each language, they demonstrate the benefit a behavior representation can have toward automated

INDEX			
SUCCESS CLAUSE			
METHODS	CONTEXT - 1 TASK-NET - 1	CONTEXT - $\cdots$ TASK-NET - $\cdots$	CONTEXT - N TASK-NET - N

Figure 3.1: Typical makeup of a RAP entity

behavior sequencing. The following sections describe some common plan execution languages implemented in various hybrid architectures, including [12,16,18,24,32,51].

3.1.1 Reactive Action Packages (RAPs). RAPs (Reactive Action Packages) [16] are described as the basic blocks for building a situation-driven execution system. A RAP is a representation that groups together all known ways to accomplish a task in various situations. The task that a RAP accomplishes acts as the index for identifying the RAP. Since the RAP is a collection of methods for accomplishing a particular task, the environment at the time of RAP selection acts as the indexing method for selecting the most appropriate RAP. Therefore, the RAP selected is the one that most effectively accomplishes the task for a given environment. The three major components of a RAP are: *Index*, *Success Clause*, and the *Methods* that accomplish the task for different situations. The *index* directly corresponds to the task-goal that a particular RAP achieves. The *success clause* describes the test that indicates when the goal of the task is completed, thus indicating that the task is complete. The *methods* within the RAP describe how to accomplish the task for different environments, or situations.

Each *method* of a RAP consist of a *context* and a *task-net*. The *context* describes the test used to indicate if the *method*, given the current situation, is appropriate for accomplishing the task. If this *method* is appropriate for the current situation, the *task-net* gives the detailed steps (behavior activations) required for successfully accomplishing the task. The steps to accomplish the task are either subtasks or primitive actions that the robot can apply. Figure 3.1 shows the break-down of components for a typical RAP.

The components of RAPs enable a fixed execution algorithm within the RAP interpreter to coordinate task execution. Figure 3.2 shows the control loop for RAP execution. The RAP interpreter, which is the Sequencer in early implementations of the 3T architecture, selects each RAP in the task agenda for consideration. After querying the current state, the interpreter selects the appropriate method and either decomposes the subtasks or sends primitive actions to the robot. If the method contains subtasks, the task indices are used to select the appropriate RAP from the RAP library and this RAP (or task) is added to the task agenda. This control loop continues until the RAPs are decomposed to robot primitives and the plan, or task-goals, are met. This process allows each RAP to pursue its goals concurrently and each will only act in the appropriate environment.

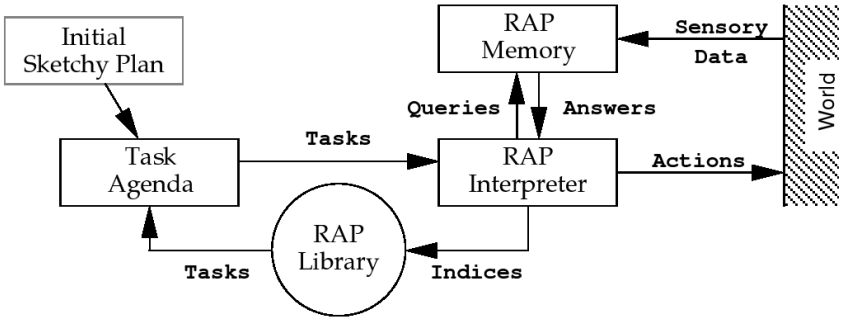


Figure 3.2: The RAP Execution System [16]

The major drawback of the RAPs system is that the programmer must know all the tasks that the system must accomplish and develop task-net descriptions to accomplish these tasks for every possible situation the system may encounter. Knowing every situation that a system encounters for each task in a dynamic environment is unrealistic. Since each RAP is hard-coded for a particular environment for accomplishing very specific tasks with limited flexibility, everytime the system environment changes, the RAPs must be reprogrammed and tested, which does not conform to the goals of robustness and modularity.

3.1.2 Reactive Plan Language (RPL). Reactive Plan Language (RPL) is a descendant of the RAP notation with the following differences [32]:

- RPL plans look like Lisp programs and thus the syntax is more recursive
- Explicit constructs for high-level concepts (e.g. interrupts and monitors)
- World state is not maintained by the interpreter

Although many Lisp programs are viewable as valid RPL plans, the intent of an RPL plan is to describe a behavior that is driven by the environment. To accomplish this, the agent's behavior is govern by temporal changes, or *fluents*. *Fluents* are conditional statements that detail a condition that must be true for an action to be executed. These fluents are controlled by state model variables or direct sensor input.

RPL "mentally executes" a plan to test if the current plan can achieve the desired goals. This is called *projection mode* in RPL. This type of internal simulation can be difficult or impossible in dynamic environments. The computation time alone may require more time than the changing environment allows. Therefore, this mode in dynamic environments is less than useful for reactive behavior planning and execution.

Like RAPs, RPL suffers from the requirement that the programmer has *a priori* knowledge of the language, system capabilities and intended environment. Changing the system capabilities or intended environment may require a complete code rewrite.

3.1.3 Executive Support Language (ESL). The Executive Support Language (ESL) is a language for encoding execution knowledge in embedded autonomous agents [18]. Gat presents ESL as an implementation substrate for the sequencing component of a TLA, such as later 3T implementations. Unlike RAPs and RPL, ESL is not intended for automated reasoning or formal analysis. It was intended to be an easy-to-use tool that is powerful and flexible. ESL uses a *cognizant failure* concept to handle errors and failures. Assuming failures are inevitable, ESL can fail and recover from errors, which includes recovering from infinite loops. The behaviors in ESL are identified by the goals that they achieve and the conditions that make the behav-

ior's method appropriate. Allowing for behaviors, or tasks, to wait for events, ESL supports multiple concurrent behavior execution.

The strength of ESL is the design for a lightweight, flexible plan execution language. However, it suffers from the detailed behavior description of the behavior's execution and the *a priori* knowledge of the working environment.

3.1.4 Procedural Reasoning System (PRS). The Procedural Reasoning System (PRS) is a high-level control and supervision language adapted to autonomous robots to represent and execute procedures, scripts, and plans in dynamic environments [24]. Ingrand *et al.* describe the PRS as the link between the high-level planning and the low-level modules. The LAAS architecture [1] uses the PRS language within its Sequencer. Like RAPs, PRS uses behavior libraries that explicitly map out low-level plans. Therefore, the Sequencer selects the appropriate library entry and executes its associated motor commands. There is no reuse of code and no actual handoff to a control layer.

The PRS provides the tools for representing and executing plans and procedures to accomplish intended goals or tasks. The world model, or the system's current belief is maintained by the PRS with derived belief's or user entered static belief's [25]. The behaviors of the robot system are described by the goals in the PRS. Knowledge for accomplishing these goals are stored in knowledge areas (KAs).

KAs are declarative procedure specifications that describe the conditions for which the KA is useful and steps for accomplishing the goals of the KA. Each KA can be viewed as a task-tree that requires certain goals to be true in order to activate the next step. Additionally, each step of a KA may activate other goals that may activate additional KAs. Therefore, facilitating concurrent execution of KAs for accomplishing multiple goals. Since one KA can activate another KA, it is feasible to anticipate a situation where one KA is negating another KA thus neither KA will ever accomplish its intended behavior or goals.

PRS must access a library of plans that satisfy all the tasks that the robot is intended to accomplish [24]. The plans in the library are not combined to create other plans and thus does not promote behavior, or plan reuse. Modular planning systems and robust constructs are ideal for multi-layer architectures and therefore requires extensions to the PRS for optimal effectiveness.

3.1.5 Proprice-Plan. In [12], Despouys and Ingrand propose an approach that couples planning with plan execution. This approach is called Proprice-Plan and integrates plan supervision and execution with different planning techniques. The authors claim that the transition between planning activities and execution are seamless. The execution model of Propice-Plan is based on PRS. Many planning techniques are implemented in Propice-Plan but for the scope of this thesis, the focus is on the execution model.

An aim of Proprice-Plan is to unify the representation for planning and execution control. The representation uses the PDDL formalism (Section 3.2.2) and is denoted as an operational-plan. The algorithms of the execution module are largely inspired by PRS but the functionality of the execution module differs in two ways. The execution module may request a new operational-plan from the planner if there are no operational-plans that currently achieve the desired task or goal. Additionally, the execution module receives and considers the recommendation of an anticipation structure that simulates the outcome of operational-plans employed in the current state. However, the system is implemented with a procedural context that limits robust and flexible execution.

The LAAS architecture uses Propice as a procedural executive to close the loop between the levels of the architecture, but Propice does not directly interpret the plan built by the temporal planner (I_XT_ET) [30]. I_XT_ET has no *a priori* knowledge about execution. Thus, a temporal executive called $I_XT_ET -_E X_EC$ was added to I_XT_ET to interleave more closely planning and temporal execution, especially to: regularly

update the plan under execution, reactively replan in case of failure, and incrementally replan upon arrival of new goals [30].

3.1.6 Reactive Model-Based Prog. Language (RMPL). The Reactive Model-Based Programming Language (RMPL) [51] is an object-oriented, constraint-based language in a similar style to Java that follows a *model-based programming* approach. A *model-based programming* approach refers to embedded programming languages that can control and reason about underlying entities from engineering models. RMPL control programs can be viewed as robot behaviors that accomplish a task or resemble a behavior. The constructs of RMPL provide behaviors with the capability for conditional branching, preemption, iteration, and concurrent and sequential composition. These constructs enable behaviors that range from simple, reactive behaviors to complex, multi-task achieving behaviors. Therefore, RMPL can offer some of the goal-directed tasking and monitoring capabilities that RAPs and ESL offer. However, RMPL constructs also offer synchronous programming.

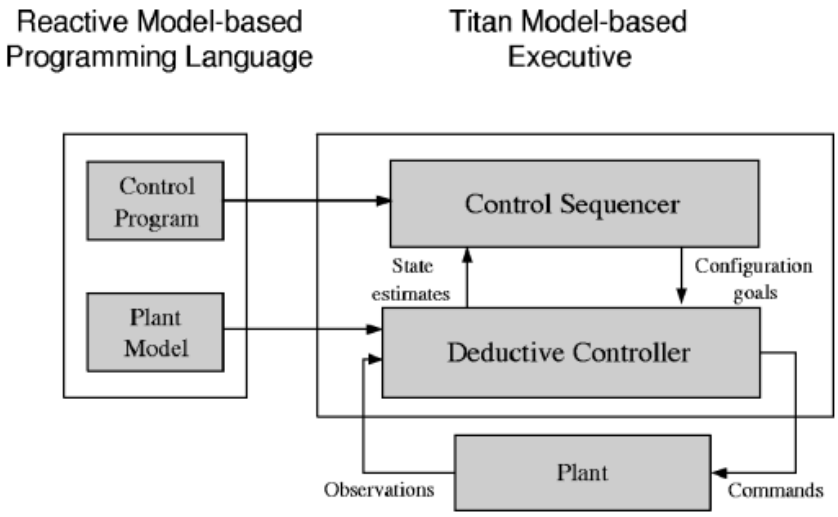


Figure 3.3: Architecture for a model-based executive using RMPL [51]

The strength of RMPL control programs is that state assignments are used as both assertions and as execution conditions. This allows for the state to contain

values that are not directly observable and promotes multiple levels of abstraction for planning and behavior execution. This language employs a dedicated executive for controlling robot behavior, named *Titan*. *Titan* uses a reactive control loop to continually monitor the current state for failures and transition from the current state to the desired configuration goals. The executive used with RMPL actually spans the Sequencer and Controller layer of the TLA paradigm (Figure 3.3). It consists of a Control Sequencer and a Deductive Controller. Many low-level reasonings, which can be coded within a behavior, are deduced and analyzed within the deductive controller. This implies that behavior descriptions are created and maintained in the sequencing layer. This convention does not promote a reactive sensor-to-action pair behavior implementation since the behavior is decomposed further in the Controller. Furthermore, the decomposition of the behavior model may limit a behavior's reactivity since it must be decomposed every cycle.

3.1.7 Common Weaknesses. Plan execution languages all suffer from similar weaknesses. The primary weakness is that these languages do not follow modular software coding practices and rely heavily on language understanding and syntax. The behaviors and the associated planning are programmed using language constructs. This dictates behavior implementation where the description of the behavior, through language constructs, is its implementation rather than the description being an abstract representation that does not restrict the implementation. Therefore, systems with task-control languages cannot change a behavior's implementation without changing its description, which causes a change in the planning components. This suggests an inability for software component reused within different robot systems or environments. These languages all have their own specific syntax and control semantics that require the programmer to know how the system functions as a whole and dictates how behaviors are described, implemented and planned. Additionally, behaviors, plans or procedures must be painstakingly detailed to include all environment conditions and all possible system tasks. Programming behaviors then becomes less intuitive and

more about fitting it to a language that requires an intimate knowledge of its syntax and semantics to create any behavior. Conversely, this thesis proposes (Research Goal 2) that developing a representation for a behavior and describing a behavior within an object lends itself to a formal selection process that can be analyzed and robustly implemented.

Another weakness of these languages is their tight coupling and interleaving of the sequencing and control layer's functional components. Since these languages are most appropriately implemented in the sequencing layer of the TLA paradigm where the layering is hierarchical, a modular and robust mechanism for plan execution is ideal. However, plan execution languages often perform the functions of both the Sequencer and Controller without a clean division between the two layers. Therefore, changing the intended environment or system capability requires an almost complete rewrite of the sequencing layer. In response to this tight coupling, this thesis proposes a robust and modular architecture design with defined links that loosely couple system components and functional layers (Research Goal 1).

3.2 Planning Domain Languages

The motivation for creating a planning problem representation is to have a way of describing a problem and solving it by taking advantage of its logical structure. Action representation has been researched since the early 1970's with the introduction of the STRIPS language [15]. By creating a representation that accurately describes a behavior, the Sequencer can automate the behavior selection control loop without knowledge of the underlying implementation of the behavior. The following sections describe the evolution of planning domain languages and action representations from its introduction (STRIPS) to the current problem specification language (PDDL).

3.2.1 STRIPS. In 1971, Richard Fikes and Nils Nilsson proposed an automated planner called Stanford Research Institute Problem Solver (STRIPS) [15]. STRIPS is a goal-based planner that uses first-order predicate calculus well-formed-

formulas (WFFS) as the goals (and subgoals) within a world model represented by first-order predicate calculus. The basic element for building solutions is by using the concept of an operator. These operators are viewable as actions (or behaviors) of how the robot functions in an environment and how these actions affect the world state. The aim of the planning system is to generate a sequence of operators that will affect the environment to a desired world state. In order to generate these plans, the operators must be described, or represented, in a way that can be searched.

STRIPS operators are described in three major components: Name of operator and its parameters, preconditions, and effects. the first component is simply the name of the operator an the parameters that the routine accepts. These parameters are objects that the operator affects (for example, boxes that are moved) or just a constant value used by the operator's routine (for example, the desired move location). The preconditions component is a formula in first-order logic. The precondition formula must be a theorem in the world model for the operator to be applicable, otherwise the operator cannot be activated. The effects component defines the effects on a set of WFFS after the application of the operator. More generally, the preconditions are the list of world requirements that must be true in order for the operator to become active, and the effects component is a list of effects on the world state after the operator has completed. Since the effects can change current beliefs in a state model, like box A that was current on box B is now on box C, the effects component has a list of added beliefs and removed beliefs. The components of operators implies that a plan consists of a sequence of operators that must be run sequentially where one operator cannot begin until another one finishes. Additionally, all effects must be added in the operator's description. In dynamic and complex environments (stochastic domains), this could be impossible to determine and thus makes STRIPS insufficiently expressive for some real domains [42].

3.2.2 PDDL. Although its algorithmic approach has been less influential than the action representation, most planning systems use one variant or another from

the STRIPS language [42]. These STRIPS variants and the notable benefits of other planning formalizations have given rise to the standardized syntax called the Planning Domain Definition Language (PDDL) [42]. PDDL is the problem-specification language originally developed by the AIPS-98 Competition Committee for use in defining problem domains and has since been used in planning competitions [21]. The authors identify that PDDL supports the following syntactic features:

- Basic STRIPS-style actions
- Conditional effects
- Universal quantification over dynamic universes (i.e., object creation and destruction),
- Domain axioms over stratied theories,
- Specification of safety constraints.
- Specification of hierarchical actions composed of subactions and subgoals.
- Management of multiple problems in multiple domains using differing subsets of language features (to support sharing of domains across different planners that handle varying levels of expressiveness).

PDDL's aim was to provide an empirical comparison of planning systems and the sharing of problems and algorithms. By having a language standard within the planning community, benchmarks and system comparisons can be empirically evaluated. To accomplish this goal, PDDL contains subsets of different planning techniques. PDDL is a good tool for expressing deterministic environments but most environments are dynamic and uncertain and thus a planner may require extensions to the notation. Although the constructs of PDDL alone cannot accurately describe all concurrent behaviors that behavior-based systems implement, the existence of PDDL validates the overall community approval of the benefits for formal representation in AI planning problems.

3.3 Deliberation to Control Interfaces

This section presents past and current research of systems and techniques that try to interface deliberative plans and behavior execution. Researchers have identified that behavior selection in behavior-based architectures is usually hand-coded to fit an architecture and is amenable to change [2, 43]. Therefore, some techniques have been developed in attempt to automate this process to make the system more robust.

3.3.1 Universal Plans. Universal plans are plans that compactly represent every classical plan [44]. The Universal plan is synthesized automatically and executes the classical plan that corresponds to the environment that the agent encounters at runtime. This approach requires that each state follows a Markov assumption and that the programmer knows all of the states that the robot encounters. This leads to a hard-coded Sequencer that requires recoding and testing whenever the system or the working environment changes. System changes range from adding or removing a behavioral capability to slightly modifying an existing behavior to function differently. This technique is not appropriate for creating a robust and modular system and is representative of the common hand-coded behavior selection techniques.

3.3.2 Behavior Coordination. Behavior coordination is the mechanism of executing the behaviors that are selected for activation accordingly to accomplish the high-level goals. In [43], the authors propose a generic architecture for dynamic behavior selection, which can integrate existing behavior selection mechanisms in a unified way. Although similar to the UBF in that this mechanism addresses the problem of selecting behaviors to accomplish high-level plans, it differs from the UBF in that it dynamically changes arbitration techniques rather than dynamically selecting the behaviors and the arbitration technique. Thus, it functions more as an arbitration technique switch. This functionality is accomplished in the Controller, rather than the Sequencer, through the UBF using the arbitrated behavior hierarchy that the Sequencer provides. In [43], the hierarchies are created and implemented for specific environments or tasks and do not display the dynamic nature that we plan to

achieve in this thesis. Although the framework is modular for easily changing the behavior selection strategies, the interface that links the decisions to change the behavior selection technique is still hand-coded to the specific behaviors that the system employs.

Another approach for behavior coordination is to perform a dynamic structuring of a hybrid behavior group coordinator by using priority and Finite State Automata (FSA) [49]. Although this approach is dynamic, the majority of the work is hand-coded in the FSAs before execution. Additionally, the technique presented in [49] requires predefined FSAs as behavior group selectors (behavior hierarchies) for each task that the system accomplishes. Lastly, in [39] they use a degenerative sequencing layer that is merely a switch for activating different behavior sets (or "navigation modes"). Therefore, the deliberative layer is really controlling the robot, which is not always active.

3.3.3 Learning. Argall, *et al* [2], state that hierarchical state machines are powerful tools for controlling autonomous robots, but the control hierarchy is often hand-coded and error prone. To combat this, they use an expert learning approach to learn to select the best state machine expert to execute for a particular task and state belief. This method is beneficial for determining the best *skill* (or behavior) created for a specific task. However, defining a reward in a dynamic environment and ensuring that the system has learned the best expert for all cases is difficult. Additionally, the behavior hierarchies are still hand-coded and this technique only determines the best of the hand-coded behaviors.

3.3.4 Abstraction. Abstraction is a term that denotes some strategic form of problem simplification that involves a representational transformation [36]. By abstracting the problem space, highly complex environments can be distilled into more manageable and relevant environments suitable for a planning domain. The use of an abstraction hierarchy recursively abstracts the problem space from the original space to the most abstract problem space. This is useful for eliminating complex

details so the planning system can focus on the goal to achieve regardless of the vast initial conditions. By distilling an abstraction to a base level, high-level plans can map to base level behavior abstractions. Nourbakhsh [36] uses a state-based abstraction hierarchy and assumes action models are expressed as arbitrary mappings between knowledge states. However, the actual translation from abstract description to concrete behavior connection is not clear.

The idea that "plans should abstractly describe" the intended behavior of the system instead of strictly "dictating" it was presented in [37]. This idea contradicts the traditional model of plans as executable programs that identify precise behaviors to accomplish intended goals. In [38], Pfleger explores this concept by describing a system that produces plans that accomplish tasks but not the actual method. The method to be used is determined at "run-time". If a plan abstractly describes the intended behavior, then the abstract behavior representation can be a placeholder that is filled at run-time with the actual behavior implementation. This allows for interchangeable behaviors for dynamic updates. Pfleger refers to these plans as *plans-as-intentions* and identifies that they are not well defined and need more information for effective use in the *plan-following* phase. This thesis aims to abstract the behaviors in a well defined manner so that a plan is generated and subsequently followed. The inner workings of the behavior (or explicit implementation) are not known to the "planner" and therefore, the "planner" plans tasks to be accomplished and the actual method is determined at run-time of the behavior. This is ideal so the behavior can determine (through sensor readings) the appropriate method to accomplish its intended behavior. Actual "abstract plan" to behavior assignment is not addressed, but is suggested to occur in the *plan following* phase of an agent architecture [22].

A weakness of abstraction is the potential of generating less optimal execution plans because some underlying details are ignored. Therefore, the level of abstraction must be effective enough to facilitate planning but not broad enough to overlook crucial information. When abstracting behaviors, the higher-level planners assume that the abstract representations map directly to output with minimal uncertainty.

3.3.5 Plan Libraries. In [34], the authors argue that the performance of plan-based control critically depends on the design and implementation of plan libraries. The planning system employs the use of low-level and high-level hierarchical planning of a library of predefined low-level and high-level plans. Low-level plans are "black-boxes" that the system does not have to reason out to employ. They achieve certain goals by containing specific steps to achieving the designated goal. This poses possible problems if the abstraction of the "black box" is too high or too low. If the abstraction is too high, it may limit the behavioral functionality of the system. Conversely, if the abstraction is too low, then the time spent in the search space may outweigh the increased behavior functionality. A low-level plan's main characterization is that it substantiates the execution parameters that are abstracted away by the high-level plans. High-level plans are the "plans that the planning mechanisms have to reason about." The high-level plan contains the mechanisms for goal monitoring, failure detection and failure handling/recovery. At this high level of abstraction, the high-level goals are achieved by reasoning/decomposing the goals into sub-goals that the low-level plans can achieve. In this system, plans are expressed in RPL and thus have the same limitations described in Section 3.1.2.

3.4 Summary

When translating plans into actual robot actions and behaviors, a number of methods have been employed. Some architectures hardcode this mechanism like universal plans and some create the mechanism *a posteriori* to augment the current architecture to perform more effectively (e.g. $I_X T_E T -_E X_E C$). However different, many of the techniques share common concerns and considerations, which include translating desired goal to behavior execution, monitoring behavior performance, and quickly reacting to environment changes. The constructs of plan execution languages identify that using a standardized behavior representation enables robust and modular sequencing of plans to active behaviors. Additionally, abstraction simplifies the search and allows the translation of high-level goals to low-level actions. The abstrac-

tion of behavior functionality through its representation appears to be the limiting factor for most of the techniques. By creating a behavior representation that is robust and modular, an appropriate sequencing mechanism can be developed to link the decomposed plans of the Deliberator to the behavior execution of the Controller.

IV. Sequencer Control Logic

Hybrid robot control architectures, have separate functional components (or layers) for plans, coordination, and actions. These layers can be generalized into three composite layers based on increasing levels of abstraction and temporal complexity: A reactive feedback control mechanism (Controller), a slow deliberative planner (Deliberator), and a sequencing mechanism that connects the first two components (Sequencer) [17]. This approach promotes systems that perform well in goal-directed and dynamic environments at the expense of system complexity. In various architectures however, the connections between these layers are typically hardcoded, so changes within one layer require modifications in other layers. By creating robust connections between layers, the software becomes more maintainable and updates within layers require reduced updating. The architectural design and implementation proposed in this chapter achieve the aim of Research Goal 1 of a robust, modular architecure design that has distinct transistions between layers and components by meeting Objectives 1-3.

The majority of hybrid architectures link the Sequencer to the Controller through the use of task-level control languages [48]. These languages require that each behavior (e.g. task-net [16]) is expressed explicitly by the syntax of the language. Therefore, an intimate knowledge of the syntax and semantics of the language is required to create any behavior. Thus, the implementation is limited by the constructs of the language. Our proposed representation (Research Goal 2) is a suggestive way to describe the behavior for sequencing, but the actual implementation is left to the creativity of the behavior architect. This means that the behavior representation is an abstract interface for sequencing and not a mechanism to dictate the behavior's concrete implementation. Thus, two behaviors can have the same descriptive representation, but the implementation and effective functionality may differ drastically.

This chapter presents a descriptive representation of reactive robot behaviors and the environment that the behaviors anticipate and affect. The behavior representation promotes robustness and modularity as being a semantic suggestion rather

than a syntactical burden like that of the task-level control languages currently in use (see Section 3.1). The semantic suggestion describes "how" the behavior functions. Whereas, task-control languages restrict the implementation of the behavior to the constructs of the language (see Section 2.2.6). This chapter also describes the dynamic behavior hierarchy generation algorithm development for the sequencer control logic. The hierarchy generation algorithm uses the abstract behavior representation interface to create a robust link between the Sequencer and the Controller in a hybrid reactive control architecture (Research Goal 3). The algorithm uses the behavior representation to search and select appropriate behaviors for activation and deactivation to complete system objectives. The Sequencer implements the hierarchy generation algorithm and seamlessly links to the Controller for behavior execution. By using the hierarchy generation algorithm and behavior representation, changes to the system code for behavior addition and functional system changes is reduced to the description of the behavior and minimal changes to the state objects.

The remainder of this chapter presents the proposed TLA design that is ideal for use with the hierarchy generation logic and the behavior representation developed in this thesis. This is followed by a description of an example domain that is referenced throughout the chapter. The next section presents the proposed representations for behaviors, plan objectives, and state objects. Then, we present the dynamic hierarchy generation logic that utilizes the proposed representations to seamlessly translate from behavior planning in the Sequencer to behavior execution in the Controller. Finally, an implementation for each of the representations is presented followed by a summary of the system.

4.1 Architecture Design

Three-layer architectures (TLAs) merge deliberative *planning and reasoning* with a reactive *control unit* to accomplish complex, goal-directed tasks while quickly responding to dynamic environments. As discussed in Section 2.2, many successfully implemented architectures [5, 11, 29, 46] are analogous to the TLA paradigm. There-

fore, the TLA paradigm is the chosen architectural design paradigm for satisfying Research Goal 1 and implementing the proposed representations and hierarchy generation logic of this thesis. Research Goal 1 calls for a robust, modular architecture design and the TLA paradigm's functional layering inherently lends itself to a robust, modular design.

As stated above, TLAs have three main layers: a reactive feedback control mechanism (Controller), a slow deliberative planner (Deliberator), and a sequencing mechanism that connects the first two components (Sequencer) [17]. The general design for a TLA is shown in Figure 2.2. However, Figure 4.1 shows a more detailed view of the TLA used for implementing the behavior representation and hierarchy generation logic presented in this thesis. Figure 4.1 shows the layout of the proposed TLA design and how the layers interact with the state, robot controls, the environment (sensors), and other layers. As shown, the architecture is hierarchical and state-based where higher levels use a more abstract state data representation and focus on longer time scale effects. The state receives sensor data updates and makes the data available to the whole system. The remainder of this section provides a high-level overview of proposed TLA design and the transitions between layers and components.

4.1.1 TLA Data Flow and Transitions. Before discussing each functional component in the architecture, a general overview of the flow of data and the transition mechanisms between layers is presented. By establishing the mechanisms and flow of data, the functionality of each layer is given more context. Figure 4.1 shows a general view of how each layer interacts. Each arrow represents the transfer of information between layers. The hierarchical design reduces inter-system dependencies and couples the layers by the mechanism (or object) used for sending information between layers. Since the architecture is a goal-driven reactive architecture, all layer functionality is based on state conditions and system objectives. Therefore, the layers

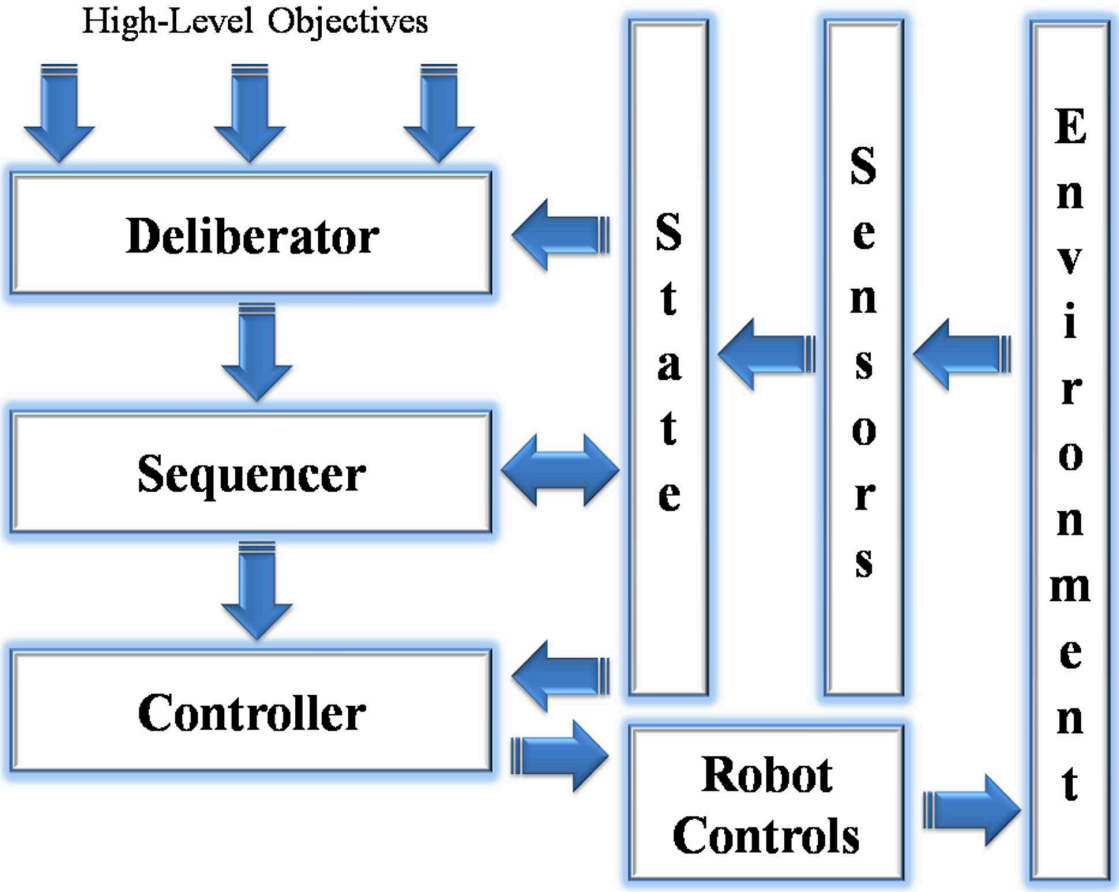


Figure 4.1: Ideal Architectural Layout of a Hybrid Reactive Control Robot Architecture for Utilizing the Proposed Representations

of the TLA either request data from the state or receive commands from one layer up in the hierarchy.

State data is an abstract representation that estimates the environment that the system occupies. The state data is made available for all the layers to interpret the data as best appropriate for that particular layer. The sensors gather data from the environment that it perceives and places this data within the state. Since the Sequencer is responsible for maintaining the abstract state representation, it is the only layer that has access to request and manipulate state data. The Sequencer uses the state to set parameters for the Deliberator and Controller to process when reacting to commands passed through the hierarchy as a result of the high-level objectives initiation.

The high-level objectives are typically sent from an outside source (e.g., a human user) and represent the desired intent of system's performance or functionality. These high-level objectives are abstract commands, such as "*Collect trash and place it into the trash bin*". The Deliberator receives the objectives and decomposes them into objectives plans (OPs). The plans are of a lower abstraction level that give more detail and ordering options and are described more thoroughly in a later Section 4.4.1. For example, "*Collect trash and place it into the trash bin*" may become a set of abstract goals that indicate that trash should be found and moved to a bin, identifies what characteristics indicate an item of trash, and where the trash bin is located. These OPs are then passed to the Sequencer. The Sequencer processes the OPs based on objectives and state conditions and generates an arbitrated hierarchy of behaviors for the Controller to execute (behavior hierarchies are described more in Section 4.4.5). Then, the Controller executes the behavior hierarchy, which generates an action recommendation. This action recommendation contains motor settings to apply to the robot controls that affect change in the environment. These changes are intended to satisfy the high-level objectives given enough time based on an abstract estimate of the environment. Table 4.1 summarizes the layer data flow and the mechanisms that couple the layers.

Transition	Mechanism
User→Deliberator	High-Level Objectives
Deliberator→Sequencer	Objectives Plan
Sequencer→Controller	Arbitrated Hierarchy of Behaviors
Controller→Robot Controls	Action Settings

Table 4.1: Data flow transitions and the mechanism used to couple the layers within a TLA

4.1.2 Controller. Layers within hybrid architectures are typically separated by abstraction and temporal complexity. Therefore, the Controller is responsible for the low-level, reactive functionality that is accomplished in real-time without knowledge of high-level goals. A software package called the Unified Behavior Framework (UBF) satisfies these criteria [52] and provides the defining line for our Controller.

The UBF is a reactive controller that abstracts the specific implementation details of specialized behaviors and permits behavior reconfiguration during execution [53]. Traditionally, a mobile robot design implements a single behavior architecture, thus binding its performance to the strengths and weaknesses of that architecture. By using the UBF as the Controller, a common interface is shared by all behaviors, leaving the higher-order planning and sequencing elements free to interchange behaviors during execution in attempts to achieve high-level goals and plans. If the controller is capable of using the behaviors in their abstract form, then it can use any concrete behavior in a uniform manner. Therefore, new behavior logic can be added without changing the Controller's implementation. By establishing behavior architectures in the context of the UBF, one can dynamically interchange between architectures capitalizing on the strengths of popular reactive-control architectures, such as Subsumption [8], Utility Fusion [41] and Colony architectures [9]. Thus, exploiting the tight coupling of sensors to actions that reactive-control architectures achieve. The common behavior interface is the establishing link between the Sequencer and the Controller by passing a behavior that represents an arbitrated behavior hierarchy that accomplishes the high-level objectives.

4.1.3 Deliberator. The Deliberator performs high-level planning and reasoning tasks [17]. This layer receives high-level, abstract requests (high-level objectives) provided by a "*higher authority agent*" that identifies the overall objectives, often a user. The Deliberator must then reason about the request and generate an OP consisting of a sequence of high-level abstract taskings that is passed down to the Sequencer. At the time of this thesis, we do not have a Deliberator in place and we therefore hand-generate the decomposed plans that the Deliberator sends to the Sequencer.

4.1.4 Sequencer. The Sequencer transforms the OPs from the Deliberator to the actions of the Controller and updates the state accordingly for re-planning. Its job is to select the behaviors that the controller uses to accomplish the objectives that are set forth by the Deliberator [17]. This requires that the Sequencer set parameters for the behaviors and change the active behaviors at strategic times to meet objectives. To do this, the Sequencer must monitor and update the state as appropriate. As seen in Figure 4.1, aside from sensors updating their data in the state, the Sequencer also stores information in the state. This allows for the setting of parameters and state variables that behaviors and other layers use. In line with the Research Goals and Objectives of this thesis, our vision for the Sequencer is of a robust software module that, after initial implementation, requires minimal software maintenance and modifications for system changes. We believe that this is accomplished by using the behavior representation (Research Goal 2) and hierarchy generation logic (Research Goal 3) proposed in this thesis. Like the behavior control logic in the UBF, the Sequencer uses the behavior representation as an abstract interface for sequencing the behaviors without knowledge of the behavior's concrete implementation. The transition from the Sequencer to the Controller is the passing of a composite behavior module that represents an arbitrated hierarchy of behaviors that, when executed, accomplish high-level objectives.

The Sequencer contains a number of components that perform specialized tasks. Figure 4.2 shows the breakout of the Sequencer's functional decomposition. These components are: Behavior Library (BL), Resource Manager (RM), Behavior Planner (BP), and Behavior Executive (BE). The next sections briefly describe the flow of information from objectives plan to arbitrated behavior hierarchy and also gives an overview of each component and its significance to the dynamic sequencing of behaviors for accomplishing complex tasks.

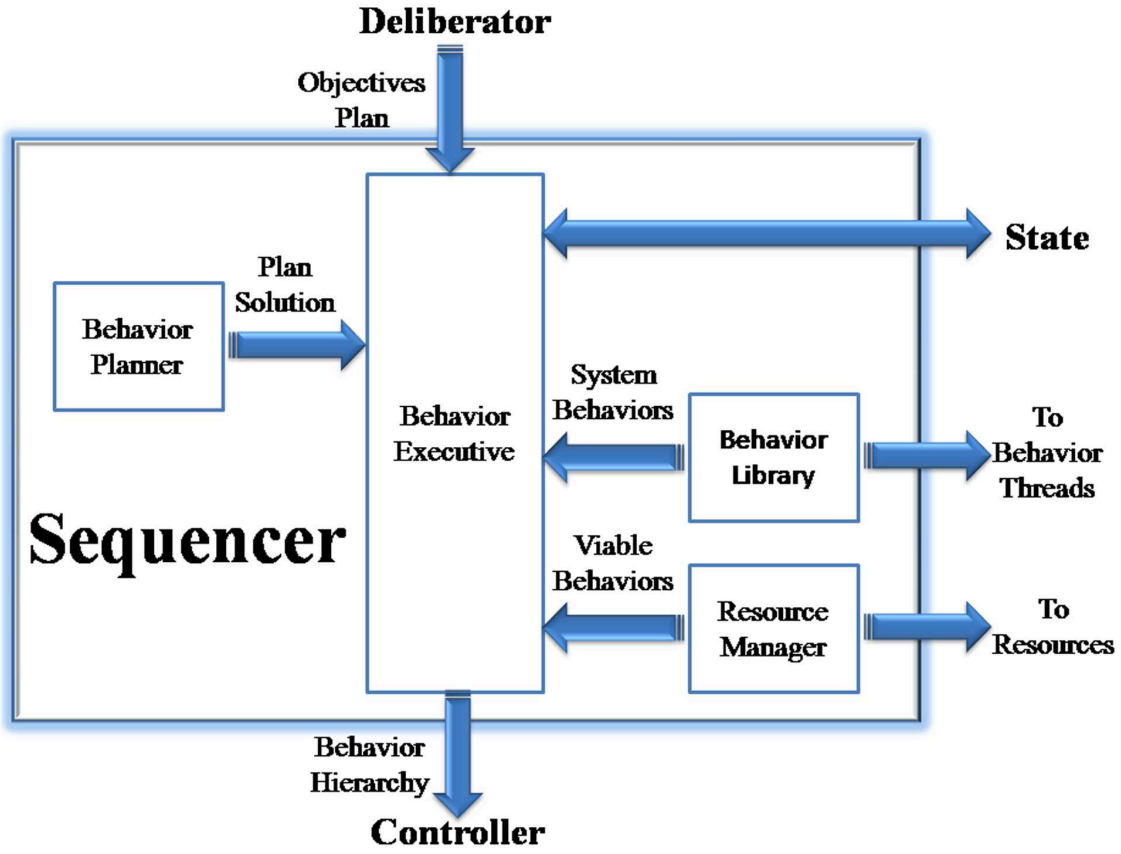


Figure 4.2: Sequencer Component Breakout

4.1.4.1 Sequencer Data Flow. The Sequencer components in Figure 4.2 perform specialized tasks that aid in translating the OPs to a behavior hierarchy that is passed to the controller. Like in Figure 4.1, the arrows represent information that is provided from one component to another. As seen, the coordinating component of the Sequencer is the Behavior Executive where it requests information

or services from the other components required for generating the behavior hierarchy. In conjunction with Figure 4.2, the timing diagram in Figure 4.3 illustrates the flow of data in translating an OP to the arbitrated hierarchy of behaviors that is sent to the Controller for execution.

When the Behavior Executive (BE) receives the OP from the Deliberator, it enters a hierarchy generation loop for translating the OP to an arbitrated hierarchy of behaviors that, when executed, accomplish the objectives set forth by the Deliberator. The BE requests a set of available system behavior representations from the Behavior Library (BL). After receiving the available behavior representations, the BE requests an analysis of the available behaviors from the Resource Manager (RM). The RM responds to the request with a subset of the behaviors that require resources (or data) that the system can currently supply. This subset is the basis of all searches performed for behavior planning, which ensures only behaviors that have its required data available are selected for activation. The BE then requests a preprocessing of the OP with the viable behavior set from the RM. The Behavior Planner (BP) returns an unsolved partial order plan that contains the core set of behaviors and ordering constraints required to accomplish the abstract goals of the OP. When the BE receives a valid partial plan, it sends another request to the BP to solve the partial plan using the set of viable behaviors from the RM. The BP solves the partial plan, if possible, and returns a solution that consists of the set of behaviors required for activation and the ordering constraints. Using the partial plan solution the BE constructs the arbitrated hierarchy of behaviors and sends it to the Controller for execution. It also sends the hierarchy to the BL and RM. The BL activates the behaviors of the hierarchy and deactivates the remaining behaviors of the library. And, the RM uses the current behavior hierarchy to manage the resource that the current hierarchy requires.

4.1.4.2 Behavior Library. The Behavior Library component is responsible for maintaining the concrete implementations and associated representations of

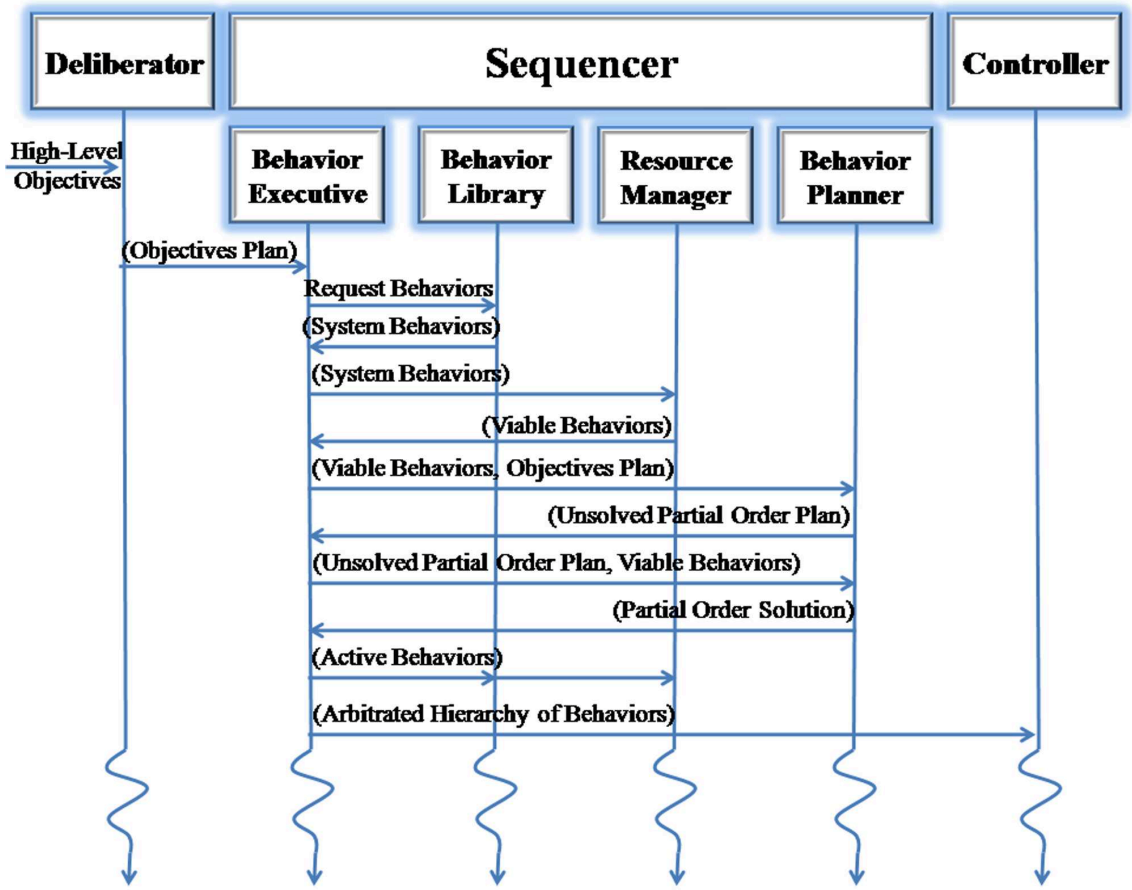


Figure 4.3: Timing diagram for the translation of an objectives plan to the arbitrated hierarchy of behaviors that is sent to the Controller for execution. An object sent to a component is identified within the parenthesis. For example, the Behavior Executive sends *System Behaviors* to the Resource Manager and receives *Viable Behaviors* as a response.

all the behaviors implemented within the system. The library initializes these behaviors and supplies the set of available behavior representations to the Sequencer. By making this component its own separate entity, the knowledge of the addition or removal of a specific behavior is contained within this component and is transparent to the other components. A system's fundamental capabilities are based upon the behaviors that are implemented within the system. Therefore, the Behavior Library component creates a system that, when its fundamental capabilities change, requires minimal change throughout the system to adapt to the change (Research Goal 1). For these changes to be transparent to the rest of the system, the behaviors must be described in an abstract representation that other components can use in a uniform manner (Research Goal 2).

Other then being a storage device for containing the set of available behaviors, the Behavior Library manages the real-time behavior threads that activate and deactivate the behaviors as dictated by the arbitrated hierarchy of behaviors. Its management capabilities maximize the real-time aspects of the behaviors while supplying the representations of these behaviors to components that are at a higher temporal complexity than the real-time behavior thread of execution. Therefore, the Behavior Library provides the mechanism for connecting the time-independent representation to the real-time execution of the behavior thread.

4.1.4.3 Resource Manager. Management of the system's hardware resources is accomplished through the Resource Manager (RM). This component monitors system resources and optimizes their use based on current tasks, planned objectives, and active behaviors [14]. The Resource Manager also answers queries about resource availability and the prospects of a behavior's activation based on its resource requirements defined in its behavior representation. The utilization of this functionality is discussed more in Section 4.4.2.

4.1.4.4 Behavior Planner. The Behavior Planner generates a set of behaviors that possibly satisfy a desired set of goals (i.e., the OP). This component

uses the behavior representation to systematically generate plans that describe the behaviors and the ordering constraints for accomplishing the requested goals. These plans consist of a set of behaviors and the ordering constraints necessary to accomplish the objectives of the OP. This functionality is currently implemented using a simplified version of RePOP (Reviving Partial Order Planning) [35], which is a variant of Partial-Order-Planning (POP). However, by creating a modular design, this component can employ any planning technique modified to generate a set of behaviors and a set of ordering constraints. Since the Behavior Executive is generating a hierarchy of behaviors based on the planners solution, a partial plan is efficient for describing behavior interactions as opposed to total order plans.

4.1.4.5 Behavior Executive. The Behavior Executive is the link from the high-level goals of the Deliberator (the OP) to the behavior execution of the Controller. As shown int Figures 4.3 and 4.2, it receives the the OP from the Deliberator and translates them into arbitrated behavior hierarchies based and their description and system availabilities. The Behavior Executive uses the behaviors supplied by the Behavior Library and queries the Resource Manager to reduce these behaviors to a set that accomplishes their assigned tasks based on the availability of required resources. It then sends the behaviors and the initial plan to the Behavior Planner to generate a core set of behaviors and ordering constraints that accomplish the desired goals if activated. After receiving a sequential behavior solution to the initial plan, it creates an arbitrated hierarchy of the solution behaviors that activates/deactivates the appropriate behaviors, at the appropriate times, to achieve the desired functionality. Finally, it sends the arbitrated hierarchy of behaviors to the Controller for the Controller to execute, to the Behavior Library for the activation and deactivation of the real-time behavior execution threads, and to the RM for the management of required resources. The hierarchy generation algorithm, which is described in more detail in Section 4.4, is implemented within the Behavior Executive and creates the defining, physical link from the Sequencer to the Controller. The link is the passing

of an arbitrated behavior hierarchy that is an executable behavior composite that the Controller uses to apply reactive control (see Table 4.1).

4.1.5 Discussion. Since integrating individual components into complete systems is ideal for making robust, modular systems [20], the architecture design described in this section utilizes the system-of-systems approach. By decomposing the layers and components to specific functional properties, the system as a whole becomes more modular. When modifications are made within one component, the other components require minimal, to no, modifications. This reduces maintenance times and increases upgrade opportunities. The layers and components within this architecture can be implemented to be independent of surrounding components are coupled only by the linking mechanisms that the individual components and layers allow. Therefore, architecture design satisfies the goal set forth by Research Goal 1 by fulfilling Objectives 1 and 2 that were define in Chapter I.

4.2 Example Domain

This section describes an example domain that is referenced for the remainder of this chapter. The example domain provides a consistent and simple example for describing how the behavior representation and control algorithm are applied. Therefore, consider a mobile office janitor robot that is tasked with discovering trash and placing it into its appropriate bin. This robot system is supplied with Sonar and Laser range data, a pan-tilt Camera with image recognition capabilities, a Gripper, and a navigation system (Odometry). Although the robot originally had an Audio output, it has been removed due to hardware failure. However, the behaviors that require the Audio output are still implemented in the system. This creates a fail condition used in a later example. The robot receives high-level tasks from an outside source that describes objects that are considered trash or sets the known location of the trash bin. The robot is programmed with the seven behaviors shown in Table 4.2 and their associated representation is described in Appendix B.

#	Name	Sensors	Function
B.1	avoid-obstacle	Laser Map	Avoid obstacles. Use an optimal path to a target location, if location is given
B.2	deliver-object	Odometry	Go-to target location
B.3	get-object	Grippers Camera Laser	Pick up target object
B.4	greeting	Camera Audio	Audibly greet recognized employees
B.5	release-object	Grippers	Release held object
B.6	scan-for-trash	Camera	Look for and identify objects (i.e. trash)
B.7	wall-follow	Sonar Map	Follow walls until all areas have been seen

Table 4.2: Example Domain Behaviors. The # column indicates the reference number of the behavior's full representation provided in Appendix B.

4.3 Behavior Representation

Behaviors are developed to accomplish a specific task or goal. By creating a standardized way of abstractly describing the characteristics of a behavior, one can create a mechanism that searches and selects appropriate behavior activations and deactivations for accomplishing desired objectives. This allows for the Sequencer to be robustly implemented to handle abstract behavior descriptions in a uniform manner.

Although behaviors interact in dynamic and stochastic environments, a behavior's perceived environment can be viewed as a deterministic environment. Behaviors are not written to react to unknown stimuli, they react to know stimuli that are possible within their perceived environment and therefore can be represented as a deterministic function. By combining multiple, concurrent behaviors that react to their own specific stimuli, the appearance of stochastic functionality is achieved. Since the environment is responded to deterministically within a behavior representation, we can design the representation of behaviors around the deterministic environment conditions that it expects.

Keeping with the rule that reactive behaviors tightly couple sensing to action, a behavior's actions are described by what it senses and the effects it has on the environment. A behavior is informally defined as the set of motor commands that trigger in response to sensor readings for accomplishing a programmed task. For example, if a set of sensors indicates that there is an obstacle impeding forward motion, an `avoid-obstacle` behavior applies appropriate motor settings to steer the robot away from the obstacle without collision. Behaviors are as complex or as simple as the architect determines appropriate.

A simple behavior has one set of outputs that is triggered by one set of input conditions. A complex behavior has multiple output sets responding to multiple input sets. We refer to these input-to-output paths as *activation-paths* (APs). Each AP description must include the initial conditions that it assumes to be true/false before execution and the output conditions that it creates.

However, the Sequencer requires more information. Although the input conditions dictate specific output conditions, a behavior effects other conditions as well. Behaviors control various settings, accomplish different abstract goals, and suggest a confidence factor (votes) for its action recommendations. All of these characteristics play a role in choosing combinations of behaviors to accomplish high-level tasks and therefore must be including in an AP. Each AP represents a single functionality of the behavior and is represented as the tuple:

$$\text{AP}= \{D, G, I, O, C, v\}$$

D is list of sensor, or computed, data required for the behavior to function properly. G is the set of high-level abstract goals that the AP accomplishes. I and O are the set of input and output state conditions. C is the set of system controls that the behavior modifies. Finally, v is the vote that the behavior generates when it delivers an action recommendation for the AP. The following sections present each of these components in more detail.

For the remainder of this thesis, the displaying convention for variable/names use the following fonts and letter casing:

- `behavior-name`
- *environment-condition*
- Sensor-Data
- Abstract-Goals
- SYSTEM-CONTROLS
- `source-code`

4.3.1 Required Data (D). Since a behavior is a tight coupling of sensor readings to motor commands, D represents the set of sensor, or computed, data required for the behavior to function properly. This data includes computed data that is directly related to the environment, but not directly from a sensor. For example, a basic `avoid-obstacle` behavior implementation may require just Sonar data so that it can arbitrarily move away from an *obstacle*. Whereas, an advanced `avoid-obstacle` behavior may require Laser and Map data to avoid the *obstacle* and remain on track to a specific target location. The Map data is computed data generated from numerous sensors, where the Laser and Sonar are directly related to hardware sensors. The basic implementation has a set D with one element (Eq 4.1), and the advanced implementation has a set D with two elements (Eq 4.2). If the system cannot provide all the data constraints in D, then the APs cannot be accomplished.

$$D = \{\text{Sonar}\} \tag{4.1}$$

$$D = \{\text{Laser}, \text{Map}\} \tag{4.2}$$

4.3.2 Abstract Goals (G). When behaviors are written, they are written to perform a specific function or accomplish a specific task. These high-level views of what the behavior accomplishes are represented in G. Using the example above, the basic `avoid-obstacle` behavior has one item in G (Obstacle-Avoidance),

and the advanced implementation also has one item in G but it is a different abstraction (Obstacle-Avoidance-Target). These values are used to give a high-level representation without moving to the decomposition (or abstraction) level of the output conditions in O. This allows for high-level planners to determine if there are behaviors that accomplish the intended goals. Additionally, a preprocessing of a high-level plan determines if the goals are accomplished by the available behaviors and therefore generates a partial plan without concern of the underlying implementation. If the pre-processing determines that the available behaviors cannot accomplish the goals of the high-level planner, it rejects the plan without entering a more detailed search for the plan's solution.

The Sequencer must have a way of determining concrete goals from idealistic goals. Concrete goals are goals that are actively pursued and can be verified as achieved, such as Go-To-Location. Whereas, idealistic goals are goals that are more difficult to verify and are typically met if necessary. A good example of an idealistic goal is Obstacle-Avoidance. This goal should be accomplished if an obstacle is encountered, but the Sequencer should not actively search for obstacles just to accomplish this goal. By identifying which goals are concrete and which goals are idealistic, the search for goal completion is restricted to just the concrete goals. The convention for initial conditions in the next Section eliminates the need for additional representation between concrete and idealistic goals in the set G.

4.3.3 Initial Conditions (I). The initial conditions of a behavior (I) represent the set of environment variables that, when true, generate an action recommendation and vote for the behavior's activation. Additionally, I represents the conditions required for the AP to produce the action outputs in (O), the effects to the controls of (C) and the vote v for accomplishing the abstract goals in G. For example, the `avoid-obstacle` behavior does not vote to control movement until it reads that there is an *obstacle* within its projected path. Thus, its set I consists of one element (*obstacle*). The set I is an abstract representation of the initial conditions and does not

dictate the implementation of how to detect an *obstacle*. Therefore, this representation accepts any abstraction of information and places the burden of identifying these abstract conditions on the behavior's programmer.

There are two types of initial conditions, active and passive. Active conditions are the initial conditions that are actively pursued to activate the AP. Conversely, a passive condition is a condition that causes the AP to execute but is not actively pursued for goal accomplishment. Passive conditions are typically associated with idealistic goals like Obstacle-Avoidance but can can be used for conditional activation such as Object-Released if *gripper-closed* is met.

An example of active conditions are `get-object`'s *gripper-open* and *tracking-object* conditions (see B.3). The Sequencer must actively pursue the activation of this behavior by ensuring the initial conditions are true for the behavior to activate the appropriate AP. For passive conditions, the Sequencer should ignore the initial conditions and not actively pursue meeting these conditions. An example of passive conditions for meeting an idealistic goal is `avoid-obstacle`'s initial condition of *obstacle* to meet the passive goal of Obstacle-Avoidance (see B.1). As stated above, the Sequencer should not actively pursue meeting the *obstacle* condition just to meet the goal. Another example of a passive goal, but one that is used to meet a conditional concrete goal, is `release-object`'s initial condition of *gripper-closed* to meet the concrete goal of Object-Released (see B.5). Since the goal is to release an object by opening Grippers, then the Sequencer is not required to seek out a *gripper-closed* condition by grabbing an object just to meet the *gripper-closed* condition. The Sequencer will have the behavior in place to release the object (or open the grippers) if necessary.

Since a behavior's functionality is described by the APs that it contains, the number of APs that a behavior contains dictates the complexity of the behavior. Each AP is required to contain a different set I. Therefore, a separate AP is required for each set of initial conditions that cause the behavior to generate a different output/vote. This representation allows for arbitrated behavior hierarchies to be de-

scribed as a composite behavior with multiple functionalities dependent upon different initial conditions. Therefore, this convention and representation allows the Sequencer to generated behavior hierarchies that are represented as a single composite behavior. These composites can be saved in the Behavior Library as new behaviors and used in later searches similar to a RAP [16].

A pictorial representation of the `avoid-obstacle` behavior with multiple APs is shown in Figure 4.4 and its tabular representation is shown in Behavior Representation B.1 (see Appendix B). The figure illustrates that, dependent upon the initial conditions, the behavior has different functionality. The behavior votes differently when there is an *obstacle* in the path, as opposed to, when there is an *obstacle* in the path and a *target-location-set* has been established. Also note that the figure shows both of these initial condition sets are passive. The set ordering of these APs are not arbitrary and play a part in AP selection based on initial conditions.

Priority of AP selection is based first on comparative conditions and then on set order. Consider the janitor robot's `avoid-obstacle` behavior B with two APs (Figure 4.4). These APs are activated when the passive conditions of an *obstacle* or an *obstacle* and a *target-location-set* is met (I_{A_1}={*obstacle*} and I_{A_2}={*obstacle*, *target-location-set*}). If the two condition sets are met, then they are first compared. Since I_{A_1} is a subset of I_{A_2} and I_{A_2} is more specific, then I_{A_2} is chosen. Conversely, if one was not a subset of the other, then the choice is made by its order in set B and A_1 is chosen over A_2.

4.3.4 Postconditions (O). The postconditions O represent the set of environment effects that the behavior intends to achieve. This intent is based on action recommendations for the behavior at the given initial state I. It can be viewed as the functionality of the behavior if given adequate time based on an approximation of the state and uncertain effects. The effects on the environment can add components or remove components. If the behavior is a basic `avoid-obstacle` behavior, then the postcondition adds *avoid-obstacle* but removes *obstacle*. However, if it is a

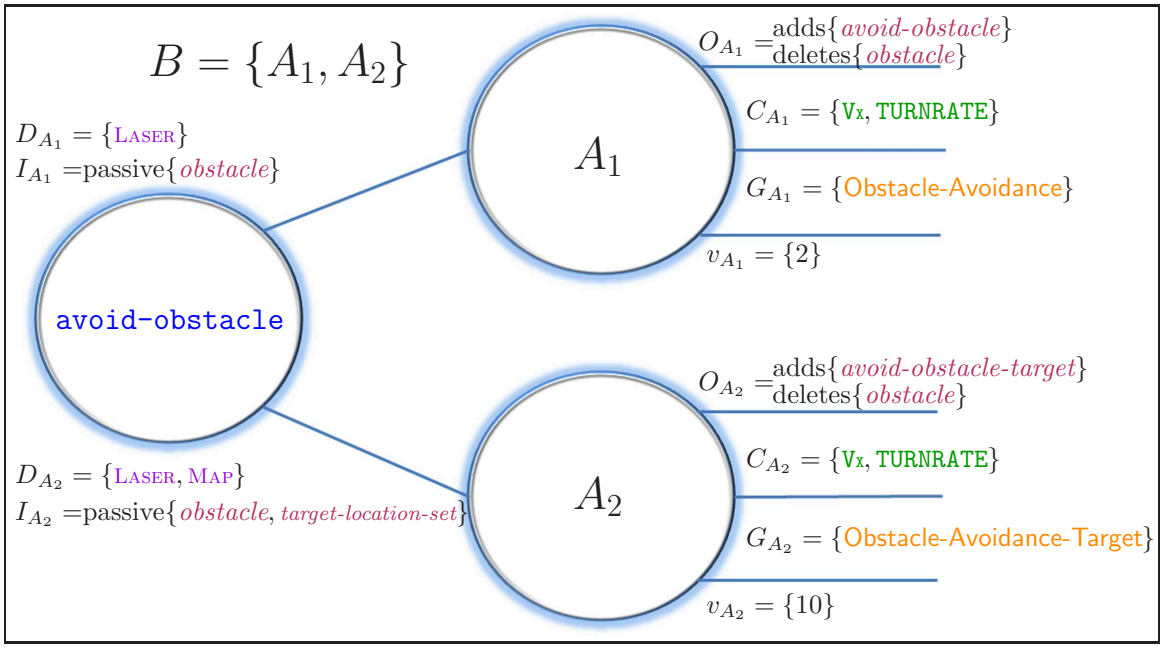


Figure 4.4: Behavior *Activation-Paths* for `avoid-obstacle`

more advanced `avoid-obstacle` behavior that finds the optimal avoidance path to reach a desired goal location, then the postconditions add *avoid-obstacle-target* but removes *obstacle*. For the latter example, Figure 4.4 shows the postconditions that are associated with each initial conditions I_{A_x} in A_x. With each of the behavior's APs, there must be an associate postconditions set that corresponds to an initial conditions set. Therefore, for each A_x in A, there is a corresponding postcondition set O_{A_x} that corresponds to I_{A_x}. As seen, with a *target-location-set*, the behavior functions as an advanced `avoid-obstacle` behavior and as a basic `avoid-obstacle` behavior otherwise.

4.3.5 Control Settings (C). Behaviors are written to affect settings for specialized controls. Most commonly, these are motor controls. A behavior potentially affects the control settings of single or multiple controls. Dependent upon which controls are set, the control loop determines the most appropriate arbiter for use with a set of behaviors. The controls that the behavior affects is denoted by the set C. This set, like postconditions O, requires values at each AP dictated by the initial

conditions set I. Figure 4.4 shows the sets of control settings for the `avoid-obstacle` behavior(C_{A_1}, C_{A_2}).

4.3.6 Votes (v). The value v in an AP represents the vote for that execution branch. Since each A_x generates an action recommendation, there must be a corresponding vote v_{A_X} for each A_x (Figure 4.4). This value is an arbitrary value that relays the strength of the action recommendation. They are used to determine the output of different arbitration techniques, which are more thoroughly discussed in Section 4.4.5.

4.4 Dynamic Behavior Hierarchy Generation

This section discusses the use of the formalized description of behaviors, Section 4.3, for dynamically selecting appropriate behaviors and arbiter hierarchies to accomplish desired objectives, which is in direct response to Research Goal 3. The Sequencer begins the hierarchy generation algorithm by searching through a library of behaviors to generate an action hierarchy package, which is a scheduling of behavior activations/deactivations that will eventually accomplish the objectives set forth by the Deliberator. Some behaviors may require activation, then deactivation and additional activations later in time. This complicates the search space since it allows cyclic branches and requires system monitoring. To make the automated link between the Sequencer and the Controller, we propose a hierarchy generation algorithm in the Sequencer that dynamically generates a behavior hierarchy. This mechanism is contained in the Behavior Executive so named for its similarity in concept to the procedural executive in the LAAS architecture, Proprice [30]. Informally, the hierarchy generation algorithm performs the following control loop:

1. Receive objectives plan (OP) from Deliberator
2. Identify behaviors that require only data that the robot can provide
3. Preprocess the OP to create a partial plan from the available behaviors that accomplish the desired high-level objectives
4. Generate a solution plan to the partial plan

5. Generate and validate an arbitration that accomplishes objectives and satisfies the solution plan
6. Generate arbitrated hierarchy of behaviors and send to controller
7. Monitor for progress, hardware changes, and new OPs

Before the hierarchy generation algorithm begins, it assumes that the Sequencer receives, from the Deliberator, a goal-set (or objectives plan) that describes the desired functionality of the system. This goal-set contains concrete and idealistic goals that suggest the ordering and priority of goal completion. Idealistic goals like Obstacle-Avoidance are included in this goal-set since situations may occur that require these goals to be accomplished or prohibit these goals from being planned. For instance, if a tasking to get the center of a room is ordered and the Resource Manager identifies that range-data either are not needed or cannot be activated. Then, a goal-set with Obstacle-Avoidance will fail to generate a behavior hierarchy since the Resource Manager will restrict any behaviors that require range data, which the behaviors that perform Obstacle-Avoidance require. Additionally, the determination or steadfast rule of what concrete or idealistic goals to achieve should not be decided within the Sequencer, but higher in the decision hierarchy, namely the Deliberator.

4.4.1 Receive Objectives Plan From Deliberator. The objectives plan (OP) that the Sequencer receives from the Deliberator contains a list of goals g that have the same value-type of goals found in the behavior representation's set of abstract goals G. The OP for the example domain is shown in Table 4.3. Each goal in the OP has an assigned sequence number and activation priority. For goals that should happen in sequence, the sequence number dictates the order (e.g. Go-To-Target happens before Release-Object). If two goals have the same sequence number, then they are expected to be accomplished before the next sequence but in no particular order. However, if there is a cause for competition, then the activation priority dictates precedence. For example, an Obstacle-Avoidance goal and Explore goal can be accomplished in the same sequence, but when an obstacle is threatening collision, the Obstacle-Avoidance goal is accomplished since it has a higher activation priority. The sequence of the OP has

an ascending precedence and the activation priority has a descending precedence. For example, Obtain-Object happens before Release-Object because Obtain-Object has a lower sequence number. Conversely, when in competition, Obstacle-Avoidance-Target is met before Go-To-Target since they have the same sequence number and Obstacle-Avoidance-Target has the higher priority value.

Goal	Sequence	Activation Priority
Obstacle-Avoidance	1	2
Target-Object	1	1
Explore	1	1
Obtain-Object	2	1
Obstacle-Avoidance-Target	3	3
Go-To-Target	3	2
Release-Object	3	1

Table 4.3: Objectives Plan for janitor robot

4.4.2 Available Behaviors . When an objectives plan enters the control loop, the hierarchy generation algorithm begins by placing the behaviors B that require only available data into a library of viable behaviors L. This process is completed everytime a new objectives plan enters the control loop. It is easy to envision a system that uses a Resource Manager that conserves energy by deactivating expensive sensors during critical times or deactivating sensors due to failure [14]. Therefore, a check of available data at every loop is necessary. This step identifies behaviors as viable if all APs, from initial condition to postcondition, are accomplished with the available data. From the example domain, the `greeting` behavior is not selected for search since the system does not have an `AUDIO` output. All other behaviors are viable.

4.4.3 Preprocess Objectives Plan. This step ensures that the objectives from the OP can be accomplished with the viable behaviors that the Resource Manager identified in L. A behavior is selected as a candidate if it contains a desired objective g from the OP in a G_{A_x}. By generating a partial plan with the ordering constraints from the OP applied to these behaviors, the sequence of goal accomplishment is

maintained. The starting state is either the current state or the projected output state of the behavior hierarchy that satisfies the OP that precedes the current OP. The end state is the outputs O of the behaviors that accomplish the goals of the last sequence in the OP. If each of the objectives in the OP cannot be assigned an accomplishing behavior, then the control loop generates a failure and jumps to step 7. Otherwise, we then search for additional behaviors that link the behavior sequences together and solve the partial plan. The behaviors and ordering constraints generated from the Op in Table 4.3 are shown in Table 4.4. As seen the ordering constraints maintain the sequence established by the OP and each behavior's set G contans a match to at least one goal g in the OP.

Behavior	OP Goal Match	Order Constaints
wall-follow	Obstacle-Avoidance	wall-follow → get-object
scan-for-trash	Target-Object	scan-for-trash → get-object
wall-follow	Explore	get-object → avoid-obstacle
get-object	Obtain-Object	get-object → deliver-object
avoid-obstacle	Obstacle-Avoidance-Target	get-object → release-object
deliver-object	Go-To-Target	
release-object	Release-Object	

Table 4.4: Preprocessed Objectives Plan for janitor robot example domain. The behaviors shown match an Achieved Goal from their behavior representation (see Appendix B to a desired goal from the OP of Table 4.3. The ordering constraints name the behavior that must accomplish its goals before (→) the next behavior accomplishes its goals.

4.4.4 Generate Solution. This step uses the preprocessed partial plan as the starting point in generating a solution. This step produces a plan with a list of behaviors and their associated ordering constraints. Since we started with the partial plan that satisfies the sequential requirements of the OP, the solution will satisfy the sequential constraints of the OP as well. However, the solution may impose more sequential restrictions on behavior activations. Since the preconditions I and the postconditions O do not represent the atomic processes that Partial-Order-

Planning (POP) expects [42], a modified planning algorithm is used to incorporate the complexity of the concurrent taskings that each behavior can encounter.

The planner uses the initial conditions to determine the conditions that must be met for behavior activation. However, if the initial condition is a passive behavior, the planner ignores the conditions and assumes that the postconditions are met when necessary. This reduces the search space but also introduces new challenges to the planner since the initial conditions are not linked to the outputs of a previous behavior, but previous outputs could dictate the passive activation. This scenario is not handled until arbiter selection and validation (step 5).

The benefits of using this planning approach allows sequential tasks to be broken into subtasks if necessary. A plan is made for the first subtask while the others are processed after subsequent subtasks complete. Each level of *sequencing priority* can be separated as a subtask using a Highest Activation arbiter [31] where the Sequencer monitors for the correct output-input pairs to advance *sequence priority* levels. Or, the sequencer can search for an arbitration that is capable of combining some, or all, of the sequential and activation priorities into one arbitrated behavior hierarchy. The difference between this planning approach and the POP approach is that POP requires a specific end plan. This plan is a set of goals that have priority and sequence numbers so that hierarchical reactive behaviors can accomplish them. Therefore, it is not simply advancing from one initial state to a specific end state. But, rather the current state to an end state that eventually accomplishes all of the intended goals in the suggested order. For simplicity, this example problem's solution plan happens to be the same as the preprocessed partial plan since all of the behavior outputs and establish orderings, is a solution to the partial plan.

4.4.5 Arbitration Selection and Validation. Arbiter selection is a crucial step. The arbiter ensures proper fusion, priority activation, and ordering of behavior action recommendations. The arbitration selection is based on two different aspects. These aspects are the controls that the behaviors affect C and how the behaviors vote

for each branch and what that branch affects. For example, the scan-for-trash behavior controls only the pan and tilt of the CAMERA-PTZ and the wall-follow behavior controls just the forward velocity (V_x) and TURNRATE and are therefore cooperative behaviors. Thus, a Utility Fusion arbiter [40] is an ideal arbiter choice for this behavior combination. Conversely, the avoid-obstacle and deliver-object behaviors both control V_x and TURNRATE and are therefore competitive, but avoid-obstacle has a higher activation priority (Table 4.3), so a Highest Activation arbiter is best for this combination (although a Utility Fusion arbiter will produce the same result for this case). An example of a final arbitrated hierarchy of behaviors for the example domain is shown in Figure 4.5. This hierarchy is of an expected output hierarchy from a complete implementation of the hierarchy generation algorithm. Since the algorithm is not completely implemented in this investigation, the hierarchy in Figure 4.5 was hand-generated to illustrate a complex hierarchy. Additionally, the modular design of the hierarchy generation algorithm allows for the actual automated generation of complex hierarchies to be a separate process. Therefore, the end result of arbitration generation is a composite behavior that represents the arbitrated hierarchy of behaviors and the actual generation process is transparent to the hierarchy generation algorithm. Automated hierarchy generation is presented as a future investigation.

The validation process of the arbitrated hierarchy of behaviors consists of ensuring the hierarchy satisfies the initial OP and the generated solution. Since a hierarchy is described as a composite behavior, it generates a behavior representation based on its output to every possible initial condition combination. By cycling through the possible environment conditions after each behavior's activation, we can generate a sequential ordering to compare to the initial plan and the generated solution. If the sequential plan satisfies the initial OP and the generated solution, then the activation priorities of the OP are validated.

To validate the activation priorities that a hierarchy establishes, the hierarchy must be traversed and the activation priorities analyzed. The initial conditions of a behavior dictate the conditions required to achieve the behavior's abstract goal

set G. Therefore, the initial conditions guide the search for determining the activation priorities for goals within the composite behavior (or hierarchy). Since the behavior that performs the function of the abstract goal performs that function when its initial conditions are met, these initial conditions are the conditions that should produce that behavior within the hierarchy. Therefore, each combination of initial conditions that accomplish all of the goals of a sequence level are searched at that sequence level. If the initial conditions for more than one abstract goal is met, then the winning goal from arbitration is given higher priority than the losing goal. If at another combination, the two goals compete again and the other goal wins, then they are given the same priority since the arbitration has no strict prioritization for these two goals. Consider the following as an example of a goal's activation prioritization within a hierarchy.

From the example domain, the `avoid-obstacle` behavior satisfies the abstract goal Obstacle-Avoidance when the initial condition *obstacle* is met and `scan-for-trash` behavior satisfies the Target-Object goal when *scanned-object* is met. If `scan-for-trash` is selected by the arbiter when *obstacle* and *scanned-object* is met, then the goal Target-Object is given a higher priority than Obstacle-Avoidance. This violates the activation priority of these two goals given in the OP (Table 4.3) and therefore generates a hierarchy failure for the plan. After a hierarchy failure, the Sequencer either attempts vote weighting or returns the hierarchy generation algorithm to step 7.

A second significant component of arbiter choice is the vote weighting of behaviors. A programmer cannot anticipate how a behavior is used throughout the life of the system. Therefore, the Sequencer requires the ability to weight the votes of each behavior within an arbiter. Although seemingly simple, this step is the final validation to ensure the behaviors' votes are correct for arbitration to perform as expected. This step attempts to trace through the arbitrated hierarchy of behaviors and adjust the weights when the goal prioritization does not perform as desired. If an appropriate weighting cannot be determined for the hierarchy to satisfy the initial OP and the generated solution, the plan is rejected. For this type of failure, either a new arbitration search is conducted or the Sequencer triggers a *plan-failure* in the

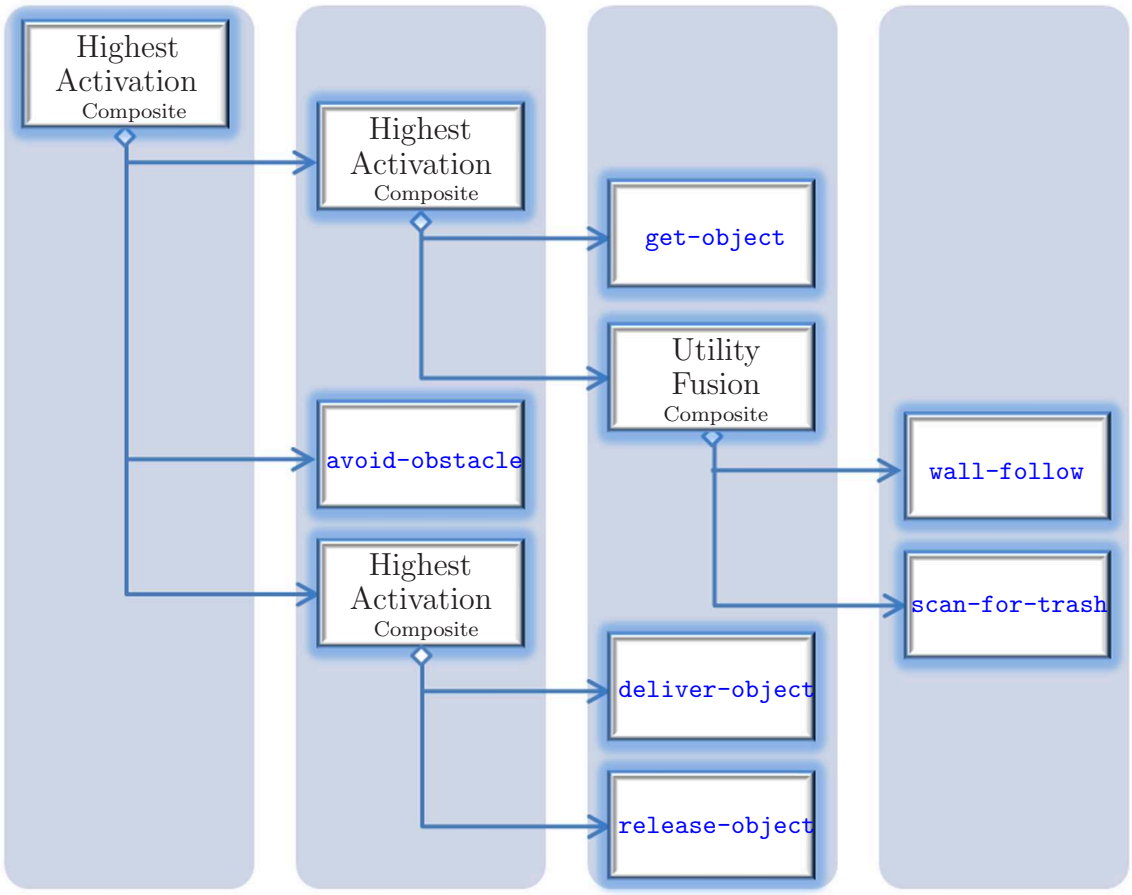


Figure 4.5: Arbitrated Hierarchy of Behaviors

state for the Deliberator to catch. Then control returns to step 7 of the control loop. The implementation of the arbiter selection and vote adjustment process is currently presented as future research.

4.4.6 Progress Monitoring. The Sequencer is charged with maintaining an abstract state representation. This includes monitoring the state for anticipated changes and dispatching the appropriate hierarchies at the appropriate times. To accomplish this, the Sequencer must have a state monitoring convention. The Reactive Plan Language uses semaphores to monitor for goal completion [32]. Although our monitoring system does not use semaphores specifically, the monitoring system waits for specific output conditions of the executing hierarchy to indicate goal completion. For instance, to determine that the initial OP is met, the outputs of the behaviors that accomplish the goals of the last sequence are monitored. In the example domain, the outputs O of `deliver-object` and `release-object` are monitored in the state to indicate that the goals of the OP (Table 4.3) have been met. However, the outputs of `avoid-obstacle` are not monitored since all of its initial conditions I are passive (Figure 4.4 and Behavior Representation B.1) indicating that its goals in G of Obstacle-Avoidance and Obstacle-Avoidance-Target are considered idealistic goals.

When monitoring for the outputs of the behaviors that accomplish the last sequence goals, the Sequencer must identify that all of the adders are present and all of the deleters are not present. For the example OP, the monitored conditions generated from the output conditions of `release-object` and `deliver-object` are:

adders: *not-have-object*, *gripper-open*, *at-target-location*
deleters: *have-object*

When these conditions are met, the Sequencer can dispatch a new hierarchy to achieve another OP or indicate achievement of the last OP and wait for a new OP. If a new OP is received, the Sequencer immediately enters the control loop at step 1, but continues monitoring for changes. When the new OP's hierarchy is generated, the hierarchy is placed into a queue to be dispatched after the monitored conditions of the previous

OP. This dispatching method allows for the correct sequential dispatching of OPs that have been decomposed into smaller sequential segments as described in Section 4.4.4.

The monitoring process also includes monitoring for conditions that potentially effect the currently planned hierarchies and solved OPs. An example of this is when the Resource Manager declares that a hardware change has occurred. Since the behaviors are initially selected based on the sensor data (or hardware) that it expects, the current (and subsequent) plans may contain behaviors that use hardware that is no longer available. This situation results in the replanning of the currently running OP and any OPs that have been processed and scheduled for later achievement.

4.4.7 Discussion. The dynamic behavior hierarchy generation algorithm is a general process for using the constructs of the behavior representation of Section 4.3 with the flexibility of the UBF [52]. It receives an objectives plan from the Deliberator and generates a sequence of arbitrated behavior hierarchies that accomplish the desired goals. By constructing the algorithm around the behavior representation, it creates a robust system that dynamically sequences behaviors based on goal requirements, resource availability, and behavior descriptions. This defined link between the Sequencer and Controller is the objective of Objective 2. Additionally, monitoring capabilities and associated handling address the objectives of Objectives 5 and 6 to handle sequential plans and environmental changes to require hierarchy regeneration. The next section discusses how the behavior representations are implemented within the architecture described in Section 4.1.

4.5 Architecture Implementation

To this point, the behavior representation and hierarchy generation algorithm have been discussed in general. In efforts toward satisfying Research Goal 1, that aims to create a robust, modular software architecture package, this section describes how the different aspects of the behavior representation and the hierarchy generation algorithm are implemented within the architecture described in Section 4.1. These

steps are the steps that must be taken to implement the proposed representation and hierarchy generation algorithm in a TLA and also the necessary considerations for modifying an existing system with new hardware capabilities and behavioral functionality. This section assumes the use of the UBF [52] as the Controller and its constructs for behavior implementation. It also assumes that the objectives plan described in Section 4.4 is supplied to the Sequencer by the Deliberator. For this investigation the Deliberator is not functional and the objectives plans are hard-coded for specific case studies.

4.5.1 Building Behaviors. Using the UBF's behavior constructs and the requirement that the behavior programmer bear the responsibility of describing the behavior, the behavior representation is contained within the behavior object. The representation is made up of the structures seen in Figure 4.6, which contains all of the requirements of the proposed behavior representation. The behavior representation is constructed of five fields: `reference`, `name`, `id`, `type`, `num_paths`, and `activationPaths`.

```
typedef struct {
    class Behavior *              reference;
    std::string                   name;
    int                           id;
    behavior_type_t               type;
    int                           num_paths;
    vector< activation_path_t >   activationPaths;
}behavior_rep_t;
```

Figure 4.6: Implemented Behavior Representation Structure

The `reference` field gives the pointer to the actual behavior object. Since the Controller's temporal complexity is at real-time calculations and the Sequencer's is at a higher then real-time complexity, we do not want to continually access the real-time object. Instead, the Behavior Planner sends the Behavior Executive a set of behavior representations that represent the instantiated system behaviors. When the hierarchy is generated, the composite behavior of the reference pointers are passed to the real-time element of the Controller.

The `name` field identifies the name of the behavior for debugging and validation of behavior execution.

The `id` field is used to identify the behavior representations without comparing the whole representation. This field is also useful for determining different instances of a behavior within a single plan. A copy of the representation can have a different id to indicate that, although the same entity, it is providing a different element in a plan. For example, one `go-to-target` reference may satisfy a Go-To-Location goal of one location, but a `go-to-target` reference with a different `id` satisfies a Go-To-Location goal of a different location.

The `type` field is used by the composite behavior object to generate a behavior representation if it encounters a composite behavior within its set of behaviors. This field is an enumeration of the different types of behavior, Figure 4.7. Since composite behaviors are dynamic in that they change properties dependent upon the the arbiter that it employs and behaviors it contains, the behavior representation must be generated whenever it is requested. If this field is `COMPOSITE_BEHAVIOR`, then the behavior must request the behavior representation to ensure that it has the most up-to-date and accurate representation for all composites.

```
enum behavior_type_t{
    LEAF_BEHAVIOR,
    COMPOSITE_BEHAVIOR
};
```

Figure 4.7: Implemented Behavior Type Identifiers

The `num_paths` field is used for efficiency and is not necessarily required to implement the behavior representation. Instead of accessing a vector object to determine the number of *activationPaths*, the querying entity can reference `num_paths` instead.

The `activationPaths` field contains all of the behavior representation components discussed in Section 4.3. This is a list of *activation-paths* that describe each function of the behavior. The number of *activation-paths* in this list must be

equal to the value in `num_paths`. Figure 4.8, shows the implemented structure of a single *activation-path* contained in the `activationPaths` field. Each *activation-path* contains one initial condition structure (`initialConditionsSet`), one post condition structure (`postConditionsSet`), one set of required data (`requiredDataSet`), one set of abstract goals (`goalsAchievedSet`), one set of action settings that it controls (`actionSettingSet`), and the vote for the action recommendation of this *activation-path*. The components of these sets are discussed more thoroughly in Section 4.5.2.

```
typedef struct{
    initial_condition_t      initialConditionsSet;
    post_condition_t         postConditionsSet;
    vector< sensor_data_t > requiredDataSet;
    vector< Goal_t>          goalsAchievedSet;
    vector< action_t >       actionSettingSet;
    int                      vote;
}activation_path_t;
```

Figure 4.8: Implemented Activation Path Structure

The structures that make up the behavior representation are the components used by the Behavior Executive to perform its intended function. By defining the behavior representation in the behavior object itself, the behavior architect has full control over the description and implementation of the behavior. Therefore, the programmer carries the burden of correctly describing and implementing the behavior so the Sequencer can assume that it is an accurate representation of estimated functionality.

4.5.2 Describing State. A system is characterized by the underlying behaviors that it possesses and how their execution is perceived [7]. Therefore, the addition or removal of behaviors within a system is what dictates its capabilities. As stated in Section 4.3, behaviors perceives only the environment that it expects. Thus, the state is represented only by these perceived values and conditions. When new behaviors are added, the state representation must be updated to incorporate any new perceptions of the environment and physical capabilities.

The initial and post conditions of an *activation-path* are implemented as two separate structures. Figure 4.9 shows that these structures have sets of active and passive conditions for the initial conditions, and adders and deleters for the post conditions. The state conditions that the initial and post conditions describe are of a state type `env_t`. This is an enumeration that defines different state conditions within the state. The behavior that uses these as conditions must identify the conditions in the same terms that the state uses. For example, if the state identifies that there is an *object* within the robot's path if the *object* is within X meters of the robot, then the `avoid-obstacle` behavior must also identify an *object* is in its path if the *object* is within X meters of the robot. Additionally, every behavior that uses this condition must identify it using the exact same method. This is required because the Behavior Executive queries the State to determine if these conditions are met or not met while monitoring for plan completion. The Behavior Planner also uses these values to satisfy one behavior's initial condition by matching that condition in the output of another behavior.

```
typedef struct{
    vector<env_t>  active;
    vector<env_t>  passive;
}initial_condition_t;
```

```
typedef struct{
    vector<env_t>  adders;
    vector<env_t>  deleters;
}post_condition_t;
```

Figure 4.9: Implemented Initial and Post Condition Structures

The `requiredDataSet` component of the *activation-path* is a list of `sensor_data_t` values. This data type is another enumeration that identifies the sensor, or computed, data available in the state. The Resource Manager must activate and control the resource that provides this data, but the access to its information is contained within the state [14]. If a new sensor or computed data is introduced by a new behavior, then the state must be updated with the constructs for querying the data and querying its availability.

The `actionSettingSet` component of the *activation-path* is a list of `action_t` values. This data type is also an enumeration of the possible action settings that the system has available for control. If a system obtains a new control device (such as adding a pan-tilt-zoom camera) and therefore requires a new behavior to control its settings, then these values must be updated within the `action` object. Additionally, any new control entity must have its associated control settings implemented within this object so the behavior can create appropriate action recommendations

The `goalsAchievedSet` component's implementation is discussed in the next section and requires updating outside entities if new goals are added. However, if a new behavior does not require any new `env_t`, `sensor_data_t`, `action_t`, or `Goal_t` entries, then only modifications to the Behavior Library component is needed. Since the Behavior Library is the only entity that has direct knowledge of implemented behaviors, any new behavior must be added to the Behavior Library for creation and initialization.

4.5.3 Objectives Plan. The objectives plan (OP) consists of a list of `Goal` objects. The `Goal` object consists of an abstract goal value g, its associated parameters, a sequence number, and an activation priority value. Goal values are of the data type `Goal_t`, which is an enumeration of all the abstract goals that are defined on the system. The parameters are used to specify concrete values for satisfying the goal value. Similar to how Saphira [27] sets "motion setpoints" within the state for the behaviors to identify the desired speeds, the goal parameters are set within the state for behaviors to identify the desired parameter values for goal accomplishment. For example, a `Goal` object with a goal value of Go-To-Location has parameters for the desired coordinates to reach. These parameters are used to set the parameters of the state that will aid in identifying when a goal has been met. The sequence number is used to dictate the order in which the goals are to be met. The activation priority is used to dictate which goal is to be met when two or more goals are in the same

sequence and are in contension for completion. An example of how these two values are used is presented in Section 4.4.1.

The `goalsAchievedSet` component of the *activation-path* is a list of type `Goal_t` goal items. These are the abstract goals that the behavior accomplishes phrased the same as the high-level abstractions of the objectives plans (OPs) from the Deliberator. When a new behavior is added that accomplishes a new abstract goal, the `Goal` object and state must be updated to identify the goal and supply the constructs for verifying its completion.

4.5.4 Building Arbiters. Currently, an abstract arbiter representation has not been developed and is suggested as a future investigation. However, arbitration selection is chosen by a hand-coded method that chooses between a Highest Activation arbiter [31] and a Utility Fusion arbiter [40]. The decision between which arbiter to use is based on the controls that the behaviors affect and the sequence of the OP. Since the sequence of behaviors is dictated by the OP, the behaviors that meet each sequence are separated into a highest activation arbiter. Then arbiter selection occurs within each separated highest activation arbiter. If a behavior is in competition with another behavior for affective control, then a highest activation arbiter is used. However, if the behavior has no competing behaviors for affective control, then a utility fusion arbiter is selected. If the selected hierarchy does not satisfy the initial OP and the generated solution, then the planning fails and exits the control loop. Future implementations will search for a different hierarchy until a successful hierarchy is found or all reasonable hierarchies are considered.

4.5.5 Sequencer Execution. The hierarchy generation algorithm discussed in Section 4.4 is implemented within the Behavior Executive presented in Section 4.1. The Behavior Executive is an execution thread that runs a continuous planning and monitoring loop, which executes the hierarchy generation algorithm. To ensure that it is a robust and modular component, the Behavior Executive utilizes the functionality of each of the Sequencer's components described in Section 4.1 to reduce the

knowledge of the components' underlying implementation. For instance, the Behavior Executive has no knowledge of how the Behavior Planner generates the list of behaviors and ordering constraints structure. It just requires that the Behavior Planner return this structure and assumes it to be a valid solution. Additionally, the Behavior Executive has no knowledge of the behaviors supplied by the Behavior Library or the resources available from the Resource Manager. It uses these components to return their appropriate behavior sets or plan sets and generates the arbitrated hierarchy of behaviors accordingly. The control loop continuously generates plans for received OPs and monitors the state for plan completion or indicators of possible replanning conditions (e.g. loss of sensor capability).

4.6 Summary

This chapter presented a hybrid architecture design (Research Goal 1), the proposed behavior representation (Research Goal 2), the proposed dynamic behavior hierarchy generation algorithm (Research Goal 3), and a system implementation that combines the proposed concepts (Research Goal 4). The architectural design and functional decomposition of layer components adheres to the integrity of the three layer architecture (TLA) paradigm described in [17], which meets Objective 1. Objective 3 is met by using the behavior representation as an interface for describing simple and complex behaviors. This representation allows the Behavior Executive to use it in a uniform manner as an abstract interface for accomplishing high-level taskings (Objective 2). The proposed hierarchy generation algorithm provides a robust mechanism for dynamically sequencing behaviors without knowledge of the high-level goals of the Deliberator and the low-level implementations of the behaviors. This suggests a modular component that requires minimal changes in response to system changes as required by Objective 4. The implementation described in Section 4.5 verifies the modular aspect of the system and demonstrates the minimal code changes required for adding new system capabilities. Since the changes are based on behavior

implementation, the proposed architecture can be applied to any system barring the appropriate updates are made as described above.

The architectural and component implementation described in this chapter directly aids in satisfying the Research Goals set forth in Chapter I by meeting some of the associated Research Goals. The next chapter presents case studies that further aid in satisfying the Research Goals using the Objectives as a guide for experimental trials. The experiments demonstrate the robust architectural design by accomplishing high-level taskings using the same software implementation of the architecture on systems with different capabilities and active behaviors (Objective 1, 3, and 4). They also demonstrate dynamic sequencing that occurs with new high-level tasking and in response to environmental changes, such as hardware failure (Objective 4, 5, and 6).

V. Results

Each Objective aids in accomplishing one or more of the Research Goals identified in Chapter I. Research Goal 4 combines the first three and requires a demonstrated behavior representation that enables dynamic behavior sequencing within a robust and modular autonomous mobile robot architecture design. This chapter demonstrates the accomplishment of the individual objectives stated in Section 1.2, which aid in satisfying the Research Goals, using three experiments.

The aim of these experiments is to demonstrate that the proposed architecture design, behavior representation, and hierarchy generation algorithm to meet the Objectives of this investigation and thereby satisfy the Research Goals. The Objectives as stated in Section 1.2 are:

Objective 1: *Show that the behavior representation and associated hierarchy generation algorithm can be applied in a hybrid robot architecture while adhering to the integrity of the three layer architecture (TLA) paradigm.*

Objective 2: *Show that there is a defined link between the Sequencer and Controller. This link must be an abstract mechanism that is robust and seamless.*

Objective 3: *Show that an abstract behavior representation, which does not require the knowledge of low-level implementation details, can be applied as an interface to simple, complex and concurrent behaviors.*

Objective 4: *Show that a hierarchy generation algorithm can use the behavior representation to dynamically generate an arbitrated behavior hierarchy for accomplishing desired goals without a priori knowledge of system capabilities and behavior functionalities.*

Objective 5: *Show that a sequence of plans generates appropriate behavior hierarchies that are assigned at appropriate times to accomplish complex high-level tasking.*

Objective 6: *Show that dynamic system changes, such as hardware failures, or environment conditions generate new hierarchies if necessary.*

By virtue of the architecture's implementation and successful execution of various behaviors, Objectives 1-3 are met with the Sequencer components presented in

Chapter IV. However, Objectives 4-6 are met within case studies. These three Objectives are the basis of the experiment design. The first section of this chapter presents the overall details that are consistent with each experiment followed by the actual case studies and their associated results:

Case Study I: Demonstrates how the behavior representation is used to dynamically select appropriate behaviors for task completion based on available resources and intended goals (demonstrates Objectives 1-4).

Case Study II: Shows how the proposed hierarchy generation algorithm uses the behavior representation to search and plan a sequence of behaviors. An arbitrated hierarchy of the behaviors is generated using these behaviors for accomplishing a sequence of tasks, which describes a high-level objective (demonstrates Objectives 1-5).

Case Study III Demonstrates the adaptation to situations that jeopardize current planned hierarchies that causes possible hierarchy regeneration (demonstrates Objectives 1-6).

The final section reiterates the results of the experiments as they apply to the established Objectives.

5.1 Experiment Details

These experiments utilize the Stage simulation environment [19], which simulates the physical Pioneer P2-AT8 robot with resources that we do and do not currently possess. Stage as a simulation environment provides the ability to add capabilities and remove elements that give the appearance of experimental failure due to sensor errors. Stage also offers the ability to induce system failures for producing controlled experimental environments but is limited by not having stochastic sensor and motor models.

The case studies in this chapter use a Behavior Library that contains every behavior described in Appendix A (Behavior Representations A.1-A.12). Therefore, the hierarchy generation algorithm begins processing and searching for a solution with every behavior in the library. The *behaviors* section of each experiment discusses the behaviors that are expected to be activated in the experiment, but does not indicate

that these are the only behaviors that the Behavior Executive uses to search for the appropriate hierarchies. Also, these expected behaviors do not indicate the behaviors that will be activated, just the behaviors that may be selected.

5.2 Case Study I: Dynamic Behavior Hierarchy Generation

The low-level behaviors in a robot system define the capabilities of that system. These behaviors can be complex or simple, reactive or goal-driven, or a combination of all. By generating arbitrated hierarchies of behaviors, the system utilizes the behaviors in combinations that accomplish more complex tasks then they do individually. A mechanism that handles behaviors in a uniform manner and dynamically adapts the hierarchy to the available behaviors and intended goals allows for a more robust system. Using Objective 4 as a guide, this experiment intends to:

1. Demonstrate that the proposed behavior representation and hierarchy generation algorithm are independent of system capabilities and behavior implementation.
2. Demonstrate that the software implementation can be transferred from systems with the same behaviors but different capabilities and resource availabilities without code modifications.
3. Demonstrate that different system capabilities generate different behavior hierarchies independent of low-level implementations.
4. Demonstrate that a simple tasking can generate different hierarchies based on behavior availability and resource capabilities

These intended demonstrations are shown by:

1. Using Behaviors with different low-level implementations (See Appendix A). Behaviors that require different sensors inherently have different low-level implementations.
2. Using different sensor configurations

(a) Sonar

(b) Laser

(c) Laser & Sonar

3. Using an identical Objectives Plan for all trials.

5.2.1 Stage Implementation (Environment). For this experiment, the environment is a simple room with scattered objects throughout (see Figure 5.1). The experiment robot is the robot in the bottom left corner, location (-4, -5). The objects scattered throughout the room are other, stationary robots and boxes. The objective is to get from the current location to the center of the room (0, 0) without hitting any obstacles.

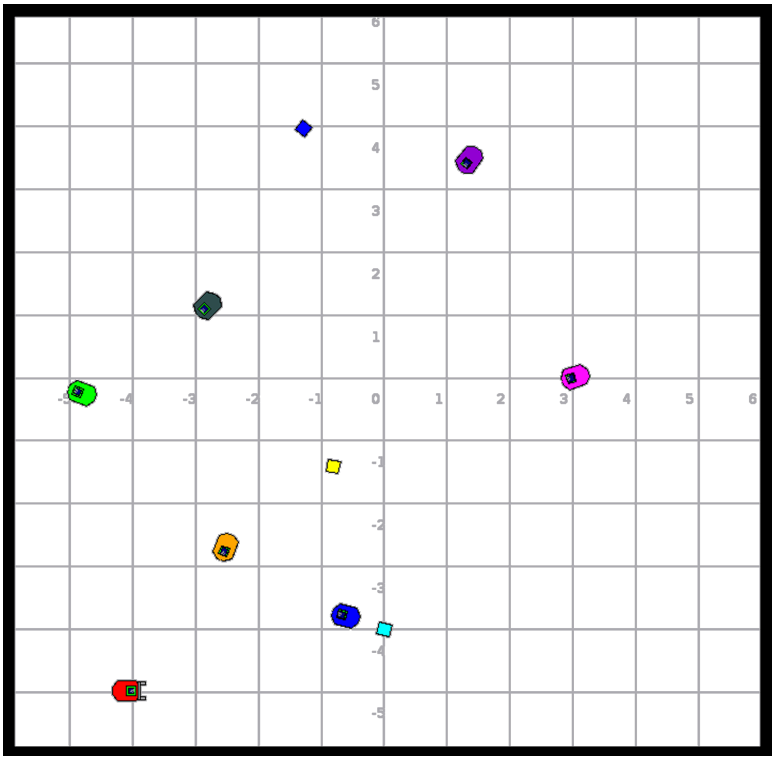


Figure 5.1: Simple Stage Environment

5.2.2 Objectives Plan. The OP for this experiment is shown in Table 5.1. This OP describes a simple task that requires the robot to get to the center of the

room (location(0, 0)) without colliding with any obstacles. The theta (θ) value refers to the angular orientation of the robot from its initial heading. Therefore, the θ parameter value of zero (0) indicates that the desired heading after reaching the goal location is equal to the initial heading at start. The intent of this OP is to show how a simple tasking can generate different hierarchies based on behavior availability and resource capabilities. This OP is used for all trails of this experiment.

Goal	Sequence	Activation Priority	Parameters
Avoid-Obstacle-Target	1	2	N/A
Go-To-XYT	1	1	x=0 y=0 θ=0°

Table 5.1: Objectives Plan for getting to the center of a room without hitting any obstacles. This plan shows that the desired location is (x, y, θ)→(0, 0, 0) and Avoid-Obstacle-Target should happen before Go-To-XYT if necessary.

5.2.3 Behaviors. The behaviors selected during these trials are intended to demonstrate different implementations of the same high-level abstract goal. Although accomplishing the same abstract goal, the low-level implementation or, in this experiment's case, the resource requirements can be drastically different. This experiment is broken into three different trials with different configurations of resource availability, which are: 1) Sonar 2) Laser 3) Laser & Sonar

Each trial robot has the same architecture software package that implements the same Behavior Library, and receives the same OP, but each robot has a different resource configuration than the other two. The following behaviors are a subset of the behaviors contained in the Behavior Library. This subset contains the behaviors that are expected to be activated during one or more of the experiment trials since they satisfy the abstract goal requirements of the OP (Table 5.1). The tabular form of their representations are shown below and in Appendix A. The appendix also shows the tabular representations of the remaining behaviors implemented within the Behavior Library. These remaining behaviors are also considered for activation during the hierarchy generation process.

`sonar-around-obstacle` (Behavior Representation A.8): Avoids obstacles by determining the best path to the target location using SONAR range finders.

`laser-around-obstacle` (See Behavior Representation A.5): Avoids obstacles by determining the best path to the target location using LASER range finder.

`go-to-xyt` (Behavior Representation A.2): Travels to a desired (x, y) location and sets its heading to θ degrees from the intital heading at system start-up.

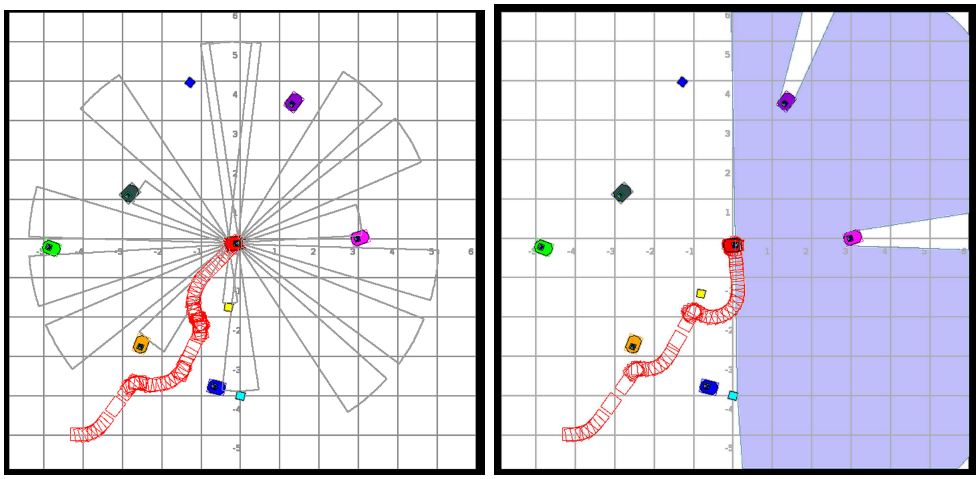
`sonar-around-obstacle`	
Initial Conditions:	
Active:	
Passive:	*treshold-min*
Post Conditions:	
Adders:	*avoid-obstacle-target*
Deleters:	*treshold-min*
Required Data:	SONAR
Goals Achieved:	Avoid-Obstacle-Target
Action Settings:	Vx
	TURNRATE
Vote:	5

`laser-around-obstacle`	
Initial Conditions:	
Active:	
Passive:	*treshold-min*
Post Conditions:	
Adders:	*avoid-obstacle-target*
Deleters:	*treshold-min*
Required Data:	LASER
Goals Achieved:	Avoid-Obstacle-Target
Action Settings:	Vx
	TURNRATE
Vote:	5

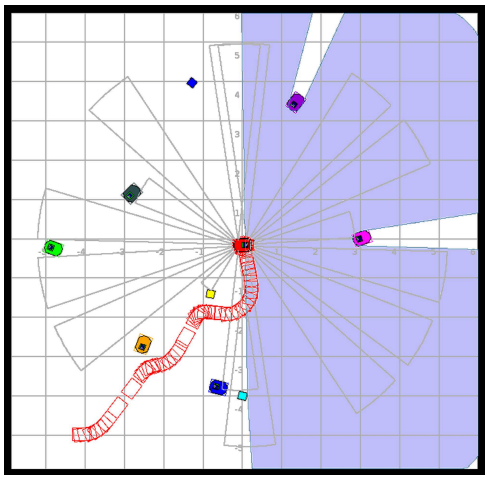
`go-to-xyt`	
Initial Conditions:	
Active:	
Passive:	
Post Conditions:	
Adders:	*target-x-location*
	target-y-location
	target-t-location
	all-stop
Deleters:	
Required Data:	
Goals Achieved:	Go-To-XYT
Action Settings:	Vx
	TURNRATE
Vote:	1

5.2.4 Results. In this experiment, three different robots with identical software and behavior libraries, but different resource availability, generated behavior hierarchies that accomplished the objectives of the same OP. The three resource availability configurations consists of: 1) only SONAR range finders available 2) only LASER

range finders available 3) both, LASER and SONAR range finders available. Figure 5.2 shows that each robot's execution of the generated behavior hierarchy accomplished the objectives of the OP with the appropriate sensors active. The 360° segmented scans in (a) and (c) indicate that the SONAR range finder is active, and the 180° wide scan shown in (b) and (c) indicates that the LASER range finder is active.



(d) SONAR (e) LASER



(f) SONAR & LASER

Figure 5.2: Resultant paths of the executed hierarchies generated for the OP in Table 5.1 with system configurations of a) SONAR range finder b) LASER range finder, and c) both range finders.

Although the implemented behaviors accomplished the task of the OP, the intent of this experiment is not to show behavior quality but to show the dynamic sequencing

of behaviors through behavior selection and hierarchy generation. Therefore, the associated behavior hierarchy for each run is shown in Figure 5.3. This figure shows that each system configuration generated a different behavior hierarchy based on behavior description, resource availability, and system objectives.

Highest Activation
go-to-xyt
sonar-around-obstacle

(a) SONAR

Highest Activation
go-to-xyt
laser-around-obstacle

(b) LASER

Highest Activation
go-to-xyt
sonar-around-obstacle
laser-around-obstacle

(c) SONAR & LASER

Figure 5.3: Resultant Case Study I behavior hierarchies for a) the SONAR range finder b) the LASER range finder, and c) both range finders.

5.2.5 Discussion. The ability to sequence behaviors for dynamically generating behavior hierarchies based on the description of the behaviors and the system state (available resources) demonstrates the robustness of the behavior representation. By creating a description that incorporates high-level abstract goals and abstract system requirements, each robot configuration was capable of completing the high-level task with different behavior hierarchies. Although a simple OP, this shows that the behavior representation is adequate for solving simple taskings that are based on abstract goals alone. If given an OP that generates the activation of behaviors that require unavailable resources, the behavior sequencing fails and no hierarchy is generated. Since this is an unremarkable trial run, we did not include it in this experiment. This experiment is a proof of concept for finding behaviors that are capable of accomplishing the desired abstract goals. However, the next experiment incorporates more elements of the behavior representation and more behavior planning components of the hierarchy generation algorithm.

5.3 Case Study II: Dynamic Sequencing with Sequential Plans

Modern autonomous robots perform more than one specific task. In the birth of hybrid robots they performed specialized tasks such as museum tour guides [50], mail delivery robots [47], or errand running robots [5]. The demand for more complex and versatile robots require accomplishment of tasks that can be reactive or goal driven, sequential or concurrent, simple or complex. Therefore, these systems must be adaptive to the environment and the tasks they are expected to accomplish. They must also be capable of processing and accomplishing series of tasks to satisfy one, or many, high-level objectives.

This experiment intends to meet Objective 5 by demonstrating the ability of the implemented hierarchy generation algorithm described in Section 4.4 to generate hierarchies for a number of sequential OPs to accomplish a complex, multi-stage task. The order of events that aid in this demonstration are that the robot:

1. Receives multiple OPs in the sequence they are expected to be accomplished.
2. Generates behavior hierarchies for each OP and places them in a dispatch queue.
3. Executes each hierarchy when previous monitor conditions are met.
4. Accomplishes the high-level task of collecting a trash item and delivering it to the appropriate location.

5.3.1 Stage Implementation (Environment). The simulated environment for this experiment is shown in Figure 5.4. The figure displays an environment where the yellow boxes (trash) are scattered around the rooms with other colored boxes (not trash). The green trash bin icon in the bottom right corner is the designated area for delivering the the yellow boxes (trash). The high-level objective for this experiment is to:

1. Explore the rooms in search of trash
2. When trash is found, pick it up
3. Bring trash to the designated disposal area

This experiment is similar to the example domain described in Section 4.2, but uses different behaviors that help to demonstrate the behavior planning capabilities of the system.

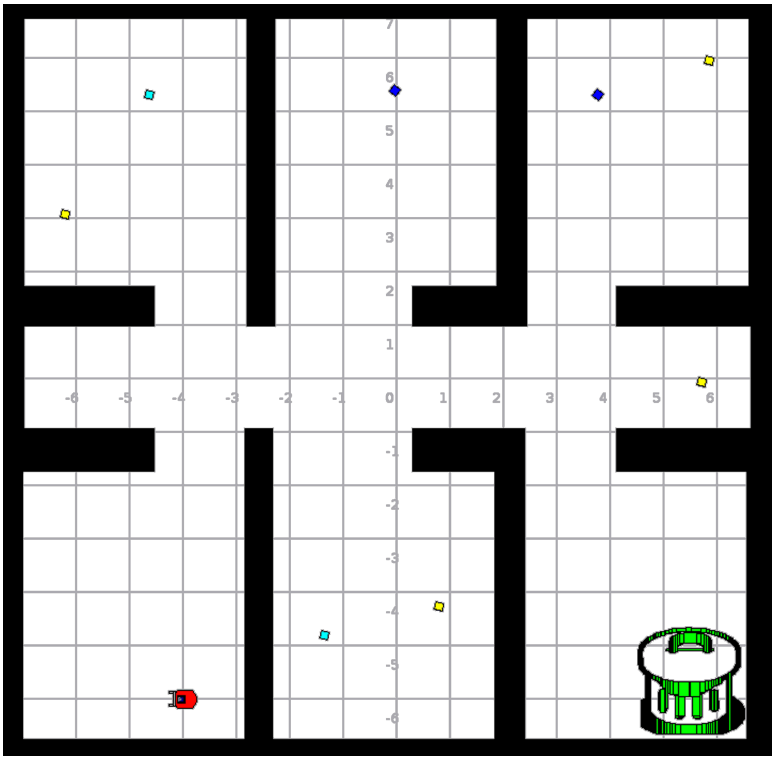


Figure 5.4: Stage environment for picking up yellow boxes and bringing them to the trash bin area (indicated by trash bin icon in the bottom right corner)

5.3.2 Objectives Plans. To accomplish the high level tasking of a janitor robot, a series of OPs must be described and sent to the Behavior Executive. The task consists of finding trash, picking it up, and bringing it to the designated trash area. Figure 5.5 shows the OPs that describe this tasking. Each OP describes a step in the janitor process:

(a) Represents the task of searching for trash while avoiding collisions and picking up the trash item when found

(b) Represents a path planning step to get the robot to the center of the map

(c) Represents the task of delivering the trash to the designated area and positioning for disposal

(d) Represents the task of releasing into the bin

These OPs represent one cycle of finding a piece of trash and disposing of it. When these OPs are repeatedly sent to the Behavior Executive, the end result should be that every piece of identified trash is placed in the designated area.

Goal	Sequence	Activation Priority	Parameters
Explore	1	1	N/A
Visual-Track-Object	1	1	Yellow
Grab-Object	2	1	N/A

(a) Find & Get

Goal	Sequence	Activation Priority	Parameters
Avoid-Obstacle-Target	1	2	N/A
Go-To-XY	1	1	x = 0.0 y = 0.0

(b) Path Planning

Goal	Sequence	Activation Priority	Parameters
Avoid-Obstacle-Target	1	2	N/A
Go-To-XYT	1	1	x = 5.5 y = -5.5 θ = 0.0°

(c) Deliver

Goal	Sequence	Activation Priority	Parameters
Release-Object	1	1	N/A

(d) Place in Bin

Figure 5.5: Objectives Plan for collecting yellow box (trash) and delivering it to the target location. a) Find a trash item and pick it up b) Move to the center of the map (path-planning step) c) Bring trash to the designated area d) drop trash into the bin

5.3.3 Behaviors. Different behaviors satisfy different requirements during dynamic sequencing. Some satisfy the initial conditions of others, while others satisfy an intended goal for the overall task. The behaviors that are selected for activation in this experiment likely satisfy both conditions at some point in the process. Therefore,

the descriptions of the behaviors selected by the Behavior Planner for activation in this experiment are shown below. The detailed representation for each of these behaviors is shown in Appendix A.

`go-to-xy`: Travels to a desired (x, y) location (see Behavior Representation A.1).

`go-to-xyt`: Travels to a desired (x, y, θ) location and sets its heading to θ degrees from the intital heading at system start-up (see Behavior Representation A.2).

`grab-object` Grabs an object with the grippers when the gripper beams are broken (see Behavior Representation A.3).

`laser-approach-object`: Slowly approaches an object using LASER until the gripper beams are broken (see Behavior Representation A.4).

`laser-around-obstacle`: Avoids obstacles by determining the best path to the target location using LASER range finder (see Behavior Representation A.5).

`release-object` Releases an object if one is held within the grippers or opens the grippers when closed (see Behavior Representation A.6).

`sonar-approach-object`: Slowly approaches an object using SONAR until the gripper beams are broken (see Behavior Representation A.7).

`sonar-around-obstacle`: Avoids obstacles by determining the best path to the target location using SONAR range finders (see Behavior Representation A.8).

`track-object` Tracks a specified target object until it is within a specified distance from the object, then it halts the robot (see Behavior Representation A.9).

`visual-track-object` Visually track a specified target object by keeping it within the camera's viewing window if possible (see Behavior Representation A.10).

`wall-follow` Travels along the walls keeping the walls (and obstacles) on its left when they are within a specified distance. Otherwise, it travels in a straight path (see Behavior Representation A.11).

5.3.4 Results. In this experiment, the OPs of Figure 5.5 were sent to the Behavior Executive one after another. These OPs were processed, behavior hierarchies generated, monitor conditions set, and a dispatch queue established. Figure 5.6 shows the path of the robot accomplishing the high-level janitor cleanup task. This

indicates that the overall task of finding a trash item and delivering it to the appropriate area is accomplished. However, the intent of this experiment is to show the dynamically generated hierarchies and the monitoring conditions set. These items are the identifiers of meeting Objective 5.

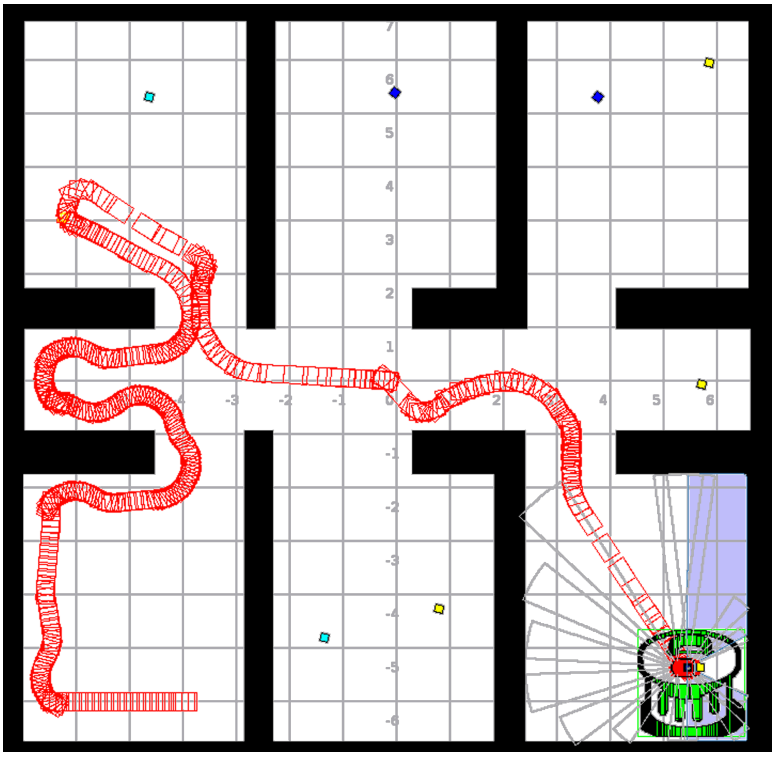


Figure 5.6: Path of the robot executing the sequential OPs from Figure 5.5 where it found a trash item and delivered it to the target "trash bin" location.

Since each run of the experiment can render different results, Figure 5.7 shows a distinct trial's hierarchies and associated monitoring conditions. When the Behavior Executive validates that a monitoring condition is met, this indicates that the next hierarchy in the dispatch queue is ready for dispatch. To meet a monitoring condition, the adders must exist in the state and the deleters must not. All other state variables are ignored. For every run of this experiment, the monitoring conditions were met and each OP was met until one yellow box was in the designated trash bin area. Sending the OPs to the Behavior Executive a second time, results in a second yellow box being delivered to the designated trash bin area. This continues until all yellow boxes are delivered to the trash bin area and the robot is left to search endlessly for more

Utility Fusion Arbiter		
Behaviors	**Monitors**	
	Adders	***Deleters***
`wall-follow`	*gripper-closed*	*gripper-open*
`visual-track-object`	*has-object*	*not-has-object*
`grab-object`		
`release-object`		
`sonar-approach-object`		
`track-object`		

(a) Find & Get

Highest Activation Arbiter		
Behaviors	**Monitors**	
	Adders	***Deleters***
`sonar-around-obstacle`	*target-x-location*	
`laser-around-obstacle`	*target-x-location*	
`go-to-xy`	*all-stop*	

(b) Path Planning

Highest Activation Arbiter		
Behaviors	**Monitors**	
	Adders	***Deleters***
`sonar-around-obstacle`	*target-x-location*	
`laser-around-obstacle`	*target-x-location*	
`go-to-xyt`	*target-t-location*	
	all-stop	

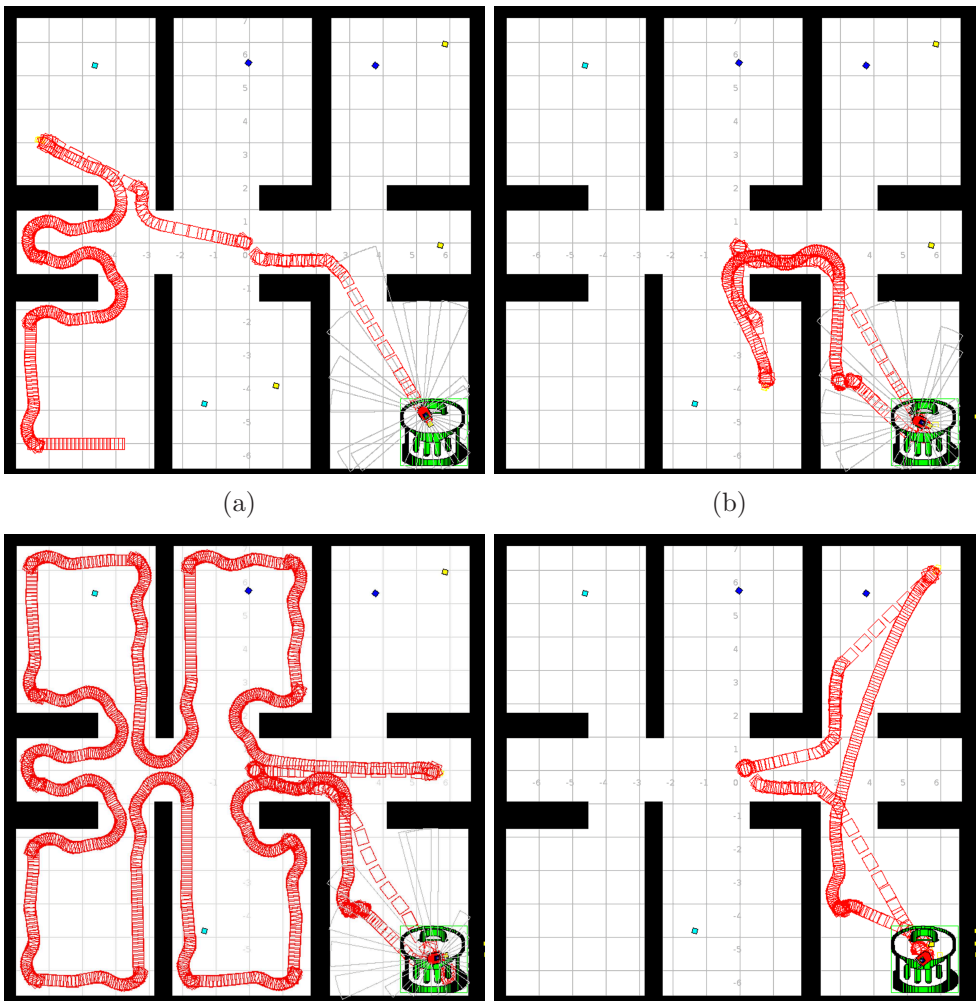
(c) Deliver

Highest Activation Arbiter		
Behaviors	**Monitors**	
	Adders	***Deleters***
`release-object`	*gripper-open*	*gripper-closed*
	not-has-object	*has-object*

(d) Place in Bin

Figure 5.7: Resultant behavior hierarchies and monitoring conditions for the sequential OPs to collect yellow boxes and place in designated area. Each hierarchy solves the associated OP from Figure 5.5 a) Find a trash item and pick it up b) Move to the center of the map (path-planning step) c) Bring trash to the designated area d) drop trash into the bin

yellow boxes. This endless search is a result of the first OP from the "*find trash and deliver it to the designated area*" tasking never meeting its monitored state, trash item targeted. Therefore, it will continue to search until it sees another yellow object. This demonstrates that the control continues to run sequential plans whenever they arrive and is shown in Figure 5.8. Note that an extra Go-To-XYT and Avoid-Obstacle-Target OP was added (not shown) to move the robot away from the trash bin, otherwise, the yellow box just placed in the bin is picked up. This extra OP is added to simulate the disposed trash as unseen and gave the opportunity to remove it from the simulation environment.



(a) (b)

(c) (d)

Figure 5.8: Collecting all (4) yellow blocks (thrash) by repeated entries of the OPs in Figure 5.5.

5.3.5 Discussion. Robot systems must be capable of combining tasks to complete high-level objectives. This requires that the Sequencer maintain a state representation to monitor progress for appropriate timing activations. This experiment demonstrates that the implemented system can receive a series of plans and execute an appropriate behavior hierarchy at the appropriate times to accomplish the series of plans in the order received. This demonstrated performance is directly in line with Objective 5.

As stated above, Figure 5.7 shows just one example of the hierarchies and associated monitoring conditions for a specific trial of this experiment. Since the hierarchy generation uses a non-deterministic selection in the planner, the hierarchies are not always identical after each processing of the same OP. This is ideal since it ensures that the same hierarchies are not the only ones exercised every time the OP is processed. This allows for other hierarchies to be explored and the possibility for logging success rates for different behavior combinations and goal accomplishments. Although, the current system does not compute these statistics, it is an option for future investigations. Additionally, the stochastic nature of the generated hierarchies demonstrates the dynamic generation capability of the Behavior Executive and the hierarchy generation algorithm, which is the goal of Research Goal 3.

The hierarchies generated by this experiment demonstrate the planning capabilities of the implemented system using the behavior representation. This meets Objectives 3 and 4 where the behavior representation is used to dynamically sequence behaviors. The behavior representation is used throughout the planning of the behaviors. For example, in order to pick up the trash object, the `grab-object` behavior requires the inner and outer beams of the gripper to be broken before activating its behavior. Therefore, the `track-object` and `sonar-approach-object` behaviors were identified by the Behavior Planner as required behavior activations to satisfy the initial conditions of the `grab-object` behavior. Additionally, the `release-object` behavior was also identified since it satisfies the *gripper-open* initial condition for `grab-object`. Occasionally, the `release-object` behavior is not identified as a constraint behavior

if the grippers are already open and the Behavior Planner selects the start state as the satisfying condition. This functionality also demonstrates the burden that is placed on the behavior programmer since the behavior's implementation is not restricted by its representation, which satisfies Research Goal 2. If the behaviors do not perform as expected or they do not evaluate the environment as expected, then the OP may never be accomplished. Therefore, a mechanism for monitoring progress failure and anomaly detection is ideal and presented as a future investigation.

This experiment explored the behavior planning and monitoring capabilities of the Behavior Executive for goal completion. However, this experiment assumed there were no possibilities of events that would cause the behavior hierarchy to be reconstructed. However, many factors can require current plans to be replanned and the Sequencer must be capable of handling these replanning conditions. One of these replanning situations is expored in the next experiment.

5.4 Case Study III: Dynamic Sequencing with Power Management

Autonomous mobile robots typically run on batteries. This places a strong need for the system to use its power resources efficiently, but still be capable of accomplishing intended objectives [14]. Therefore, a system that utilizes a power management scheme for optimizing power consumption increases system efficiency. This is accomplished in the Resource Manager (RM) of our implemented hybrid architecture. When the RM performs power management adjustments that remove a resource from availability, then previously planned OPs are no longer viable solutions. The behaviors that are assigned to each OP may require the resources that have been made unavailable. An example of this is when the RM turns the Laser off to conserve power. Then all behaviors that require Laser data will not perform as expected. Therefore, all the plans must be marked for replanning and the behavior hierarchies reconstructed. Although this is not the only condition for possible OP replanning, this scenario demonstrates the system's ability to dynamically replan the behavior hierarchies for all OPs regardless of the replanning catalyst.

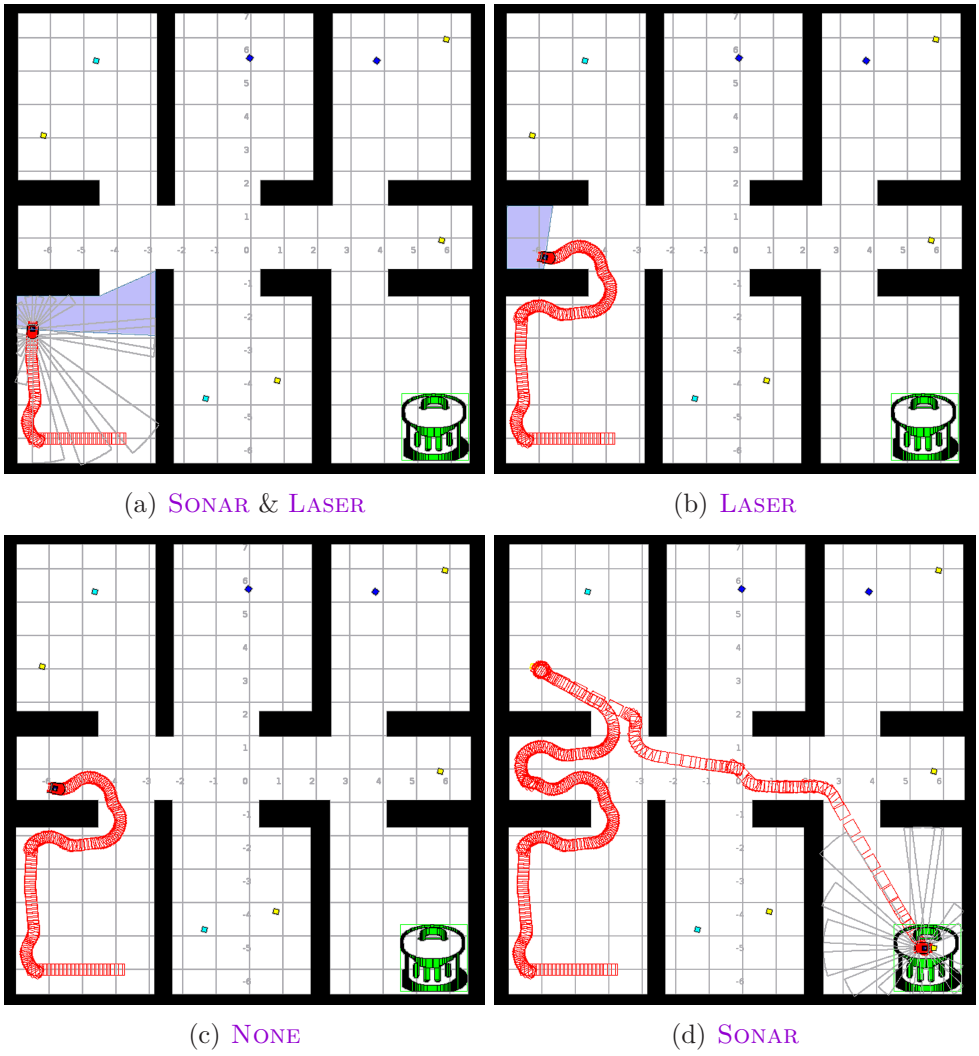
This experiment intends to demonstrate the system's ability to handle dynamic environments by replanning OPs and generating new behavior hierarchies to accomplish the OP's objectives when necessary (Objective 6). This case study uses the same environment, expected behaviors, and OPs as the experiment in Section 5.3. However, various resources are made unavailable at random intervals to observe the system's reaction.

5.4.1 Results. In this experiment, the OPs of Figure 5.5 were sent to the Behavior Executive one after another. These OPs were processed, behavior hierarchies generated, monitor conditions set, and a dispatch queue established. At random intervals, the changes of *resource availability* trigger the Sequencer to alert the Behavior Executive of the possible conflict for currently planned OPs. These random changes of resource availability occurred from resource management events within the RM. These simulated events are forced within the RM, but assumed that the RM generated the changes from management activities. The system begins with all required resource that each behavior in the Behavior Library require. The changes made to to resource availability occur in the following order:

1. Sonar made unavailable
2. Laser made unavailable (i.e. both Range Finders are unavailable)
3. Sonar becomes available (i.e. only the Sonar range finder is available)

Including the initial, system startup condition of Laser and Sonar being available, these changes produce four different resource configurations that demand hierarchy reconstruction and even plan failures. Figure 5.9 shows the environment at each change of resource availability. The 360° segmented scans in (a) and (d) indicate that the Sonar range finder is active, and the 180° wide scan shown in (a) and (b) indicate that the Laser range finder is active. The last configuration (d) shows the end result where the robot has delivered an item of trash to the designated area. However, the intent of this experiment is not to demonstrate the quality of the behaviors, but to

demonstrate the dynamic reconstruction of behavior hierarchies when environment conditions (i.e. unavailable resources) dictate.



(a) SONAR & LASER (b) LASER

(c) NONE (d) SONAR

Figure 5.9: Case Study III resultant path during resource changes with the available range finder a) SONAR & LASER, b) LASER, c) neither, and d) SONAR ranger finder.

Figure 5.10 shows the original and newly generated hierarchies after a resource change that effects the previous solutions to the OP. When possible, or necessary, the OP's are replanned with appropriate behavior hierarchies that eventually accomplish the high-level task of disposing of a trash item. Hierarchy generation fails if there are no behaviors that are capable of accomplishing a goal of the OP due to resource unavailability. This is shown to occur in (b) and (c). In (b), the current running hier-

archy for the "find & get" OP fails to generate an appropriate hierarchy and therefore halts the robot. However, the other three OPs still generated appropriate hierarchies that reflect the change in resource availability (e.g. sonar-around-obstacle is removed from the hierarchy when SONAR are unavailable). In (c), all hierarchy generations fail except for the `release-object` behavior. When hierarchy generation fails, the dispatch item for that OP is marked as a plan failure and awaits for a new replanning trigger to re-evaluate the OP. If the Behavior Executive reaches this item in the dispatch queue, then the Behavior Executive dispatches the default behavior (typically an `all-stop`) and waits until another replanning trigger occurs, or the Sequencer resets the dispatch queue and sends in new OPs. For this experiment, the robot halts when the SONAR are unavailable and when both range finders are unavailable. However, the robot returns to its task and eventually accomplishes the objective when the SONAR are made available again (Figures 5.9 and 5.10).

Sensor	Both	Laser	Neither	Sonar
Arbiter	*Utility Fusion*	*N/A*	*N/A*	*Utility Fusion*
Behaviors	wall-follow visual-track-object grab-object sonar-approach-object track-object release-object laser-approach-object	Failed	Failed	wall-follow visual-track-object grab-object sonar-approach-object track-object release-object

(a) Find & Get Transitions

Sensor	Both	Laser	Neither	Sonar
Arbiter	*Highest Activation*	*Highest Activation*	N/A	*Highest Activation*
Behaviors	go-to-xy sonar-around-obstacle laser-around-obstacle	go-to-xy laser-around-obstacle	Failed	go-to-xy sonar-around-obstacle

(b) Path Planning Transitions

Sensor	Both	Laser	Neither	Sonar
Arbiter	*Highest Activation*	*Highest Activation*	N/A	*Highest Activation*
Behaviors	go-to-xy sonar-around-obstacle laser-around-obstacle	go-to-xy laser-around-obstacle	Failed	go-to-xy sonar-around-obstacle

(c) Deliver Transitions

Sensor	Both	Laser	Neither	Sonar
Arbiter	*Highest Activation*	*Highest Activation*	*Highest Activation*	*Highest Activation*
Behaviors	release-object	release-object	release-object	release-object

(d) Place in Bin Transitions

Figure 5.10: Case Study III resultant behavior hierarchies, and effective hierarchies after resource availability reconfiguration, for the sequential OPs to collect yellow boxes and place in designated area from Figure 5.5.

5.4.2 Discussion. The ability to dynamically adapt to unpredictable environments is a driving force behind most autonomous robots. Systems that rely on hard-coded methods for performing complex task require that every possible condition is addressed. Utilizing a library of simple behaviors to generate an arbitrated behavior hierarchy to accomplish complex tasks, reduces the adverse effects of unforeseen situations. Sensors failing, behaviors not capable of meeting their objectives, and new plans arriving that augment current plans are some conditions that trigger re-evaluations of the currently planned objectives. Currently, our system only handles hardware availability reconfigurations (Objective 6) and OP sequence resets, but this experiment demonstrates the obvious capability of handling the replanning when necessary. The Behavior Executive and Behavior Planner have no knowledge of the behaviors and OPs that they are processing. These components process the Behavior representations and OPs in a uniform manner that does not rely on any specific knowledge of how the goals are met, the controls are set, the conditions are determined, etc.. Therefore, the demonstration of this experiment is the culmination of the established and implementable Objectives that satisfy the Research Goals.

5.5 Summary

The results of this chapter demonstrates the accomplishment of the Objectives described in Section 1.2. By virtue of simply performing these experiments suggests Objectives 1-3 are met. The accomplishment of high-level objectives (the OPs) in these experiments shows that the behaviors' representations are sequenced dynamically and a composite behavior that represents an arbitrated hierarchy of behaviors is passed to the Controller (Objectives 3 and 4). The passing of the composite behavior as an abstract interface provides the seamless link between the Sequencer and the Controller (Objective 2). This also shows that the behavior representation and dynamic behavior hierarchy generation algorithm are successfully implemented within a TLA architecture while adhering to the integrity of the paradigm of the TLA de-

sign (Objective 1). The individual experiments further provide evidence that the Objectives are met.

The first experiment, Case Study I, demonstrates how the behavior representation is used to select behaviors for accomplishing abstract goals through its goal description and resource requirements (Objective 4). This is the first step in planning a behavior hierarchy and is essential for selecting the core behaviors for accomplishing a complex task. If the abstract goals of the high-level tasking cannot be met by an available system behavior, then it is an indication that there is a need for new behaviors, a new high-level OP, or an available resource. This case study also demonstrates the software's robustness for similar systems that have different resource availabilities. Dependent upon the available resources, the generated hierarchy may contain different behavior activations. Without *a priori* knowledge of behavior implementations or system capabilities, the hierarchy generation algorithm is capable of generating an appropriate hierarchy for accomplishing the desired task.

The second experiment, Case Study II, expands the use of the behavior representation within the hierarchy generation algorithm. The Behavior Executive uses the remaining components of the representation to satisfy the OP by selecting (or planning) behaviors based on the required initial conditions and satisfying post conditions. This case study also demonstrates that the hierarchy generation algorithm generates appropriate behavior hierarchies that are assigned at appropriate times to accomplish the complex high-level tasking of a series of OPs (Objective 5). Therefore, displaying the ability to perform concurrent deliberative planning and reactive execution.

The final experiment, Case Study III, showcases the dynamic adaptability of the system as a whole. The experiment shows that an abstract behavior representation requires no knowledge of low-level implementation details or system capabilities. Additionally, the case study demonstrates the ability to react to environment changes that jeopardize the validity of current plan solutions (Objective 6).

VI. Conclusions

Layered, hybrid robot architectures combine deliberative planning with reactive behavior execution. This thesis demonstrates that the development of an abstract behavior representation can be used as an abstract interface that spans the functional layers of a hybrid robot architecture. Along with the representation and a three layer robot architecture design, a hierarchy generation algorithm can dynamically sequence system behaviors for accomplishing high-level tasks in a robust and modular implementation. This chapter reiterates the projected benefits of creating a robust and modular robot architecture that requires minimal code modifications when new system capabilites are added. The research conclusions drawn from the test results of Chater V are presented in Section 6.2 followed by possible future investigations toward expanding the behavior representation and dynamic behavior hierarchy generation algorithm. The final section presents the final remarks of this thesis.

6.1 Summary

The development of an abstract behavior representation is intended to provide a common interface that links the deliberative planning with the reactive execution control in a hybrid mobile robot architecture. Additionally, the representation provides the constructs for a hierarchy generation algorithm within the Sequencer to dynamically sequence behaviors at a higher abstraction level than that of the behavior's low-level implementation. This creates the defining entity that allows the Sequencer and Controller to seamlessly pass the planned behavior hierarchy between layers, but still enables the components to be robust and modular.

Autonomous robot systems are becoming more complex and more desirable for use in dynamic and unpredictable environments. These systems receive high-level taskings and attempt to perform the task using a combination of deliberative planning and reactive execution. However, most systems are designed explicitly for one purpose and thus the architectural design and implementation is very specific as well. The connections between different layers of a hybrid architecture are tightly coupled where

small changes in one layer may result in major changes in another. The proposed abstract behavior representation for describing behaviors increases the modularity of hybrid robot architectures and provides a means for a more robust functional decomposition of robot architecture design. The use of the behavior representation within the suggested hierarchy generation algorithm enables dynamic selection of an arbitrated behavior hierarchy that accomplishes high-level taskings without knowledge of the underlying behaviors' implementations. This interaction between the behavior representation and hierarchy generation algorithm creates a loose coupling, but defined link, between the planning layer and behavior execution layer of a hybrid architecture.

6.2 Research Conclusions

The Research Goals 1-4 and Objectives 1-6 identified in Chapter I aim to develop a behavior representation and dynamic behavior hierarchy generation algorithm that can be implemented within a layered hybrid robot architecture as a robust and modular software package. The results provided in Chapter V demonstrate that these Objectives have been met and therefore satisfy the Research Goals. By virtue of simply performing the experiments suggests Objectives 1-3 are met. The accomplishment of high-level objectives (the OPs) in the experiments show that the behaviors' representations are sequenced dynamically and a composite behavior that represents an arbitrated hierarchy of behaviors is passed to the Controller (Objectives 3 and 4). The passing of the composite behavior as an abstract interface provides the seamless link between the Sequencer and the Controller (Objective 2). This also shows that the behavior representation and hierarchy generation algorithm are successfully implemented within a TLA architecture while adhering to the integrity TLA design paradigm (Objective 1). The individual experiments further provide evidence that the research goals are met.

The first experiment, Case Study I, demonstrates how the behavior representation is used to select behaviors for accomplishing abstract goals through its goal

description and resource requirements (Objective 4). This is the first step in planning a behavior hierarchy and is essential for selecting the core behaviors for accomplishing a complex task. This case study also demonstrates the software's robustness for similar systems that have different resource availabilities. Dependent upon the available resources, the generated hierarchy may contain different behavior activations. Without *a priori* knowledge of behavior implementations or system capabilities, the hierarchy generation algorithm is capable of generating an appropriate hierarchy for accomplishing the desired task. The second experiment, Case Study II, expands the use of the behavior representation within the hierarchy generation algorithm. The Behavior Executive uses the remaining components of the representation to satisfy the OP by selecting (or planning) behaviors based on the required initial conditions and satisfying post conditions. This case study also demonstrates that the hierarchy generation algorithm generates appropriate behavior hierarchies that are assigned at appropriate times to accomplish the complex high-level tasking of a series of OPs (Objective 5). Therefore, displaying the ability to perform concurrent deliberative planning and reactive execution. The final experiment, Case Study III, showcases the dynamic adaptability of the system as a whole. The experiment shows that an abstract behavior representation requires no knowledge of low-level implementation details or system capabilities. Additionally, the case study demonstrates the ability to react to environment changes that jeopardize the validity of current plan solutions (Objective 6).

6.3 Future Investigation

The case studies presented in Chapter V demonstrate how the behavior representation and dynamic behavior hierarchy generation algorithm are implemented as robust, modular components within a hybrid robot architecture. However, these experiments and implementations are a proof of concept in defining an abstract behavior representation that promotes dynamic sequencing in a robust, modular software package. The representation and hierarchy generation algorithm are presented

so that future additions to either is transparent. The behavior representation and planning process described in this thesis is a modification of a representation previously presented [13]. These changes increased the ability of the planner, but required no changes to other implemented components (e.g. Behavior Executive). Therefore, some short-term future investigations are presented below that will increase the planning efficiency or overall system capabilities with minimal changes to the current modular entities.

- Create an abstract arbiter representation that can be used like the behavior representation to describe different arbiters. This allows for the hard-coded arbiter selection process in this investigation to be replaced by a mechanism that searches the available arbiters for the most appropriate arbitration of the behaviors based on ordering constraints and desired goals. Additionally, this allows more hierarchy possibilities to be searched when a hierarchy selection fails, thereby reducing planning failures. This investigation is suggested in Sections 4.4.5 and 4.5.4 with the discussion of arbiter selection ad arbiter building.
- Add the vote weighting mechanism that allows the votes of behaviors to be adjusted. This increases the usability of behaviors by searching for an appropriate weight to generate the desired output of an arbitrated behavior hierarchy, and thus increases possible hierarchy solutions. This investigation is suggested in Section 4.4.5 with the discussion of arbiter selection.
- Add multiple *activation-paths* to the planning capability so that more complex behaviors are searched. The partial order planner implemented for this thesis did not include planning for conditional capabilities and therefore could not reuse generated hierarchies since the majority of the hierarchies have multiple *activation-paths*. With this modification, the option of saving generated hierarchies for future use is capable. Multiple *activation-path* planing also leads to the next future investigation.

- Since the behavior representations are passed as copies, the Resource Manager can use this to search for activation branches that cannot be used and delete that *activation-path* so that the Behavior Executive does not search a branch that will not be activated. Even when activated, the behavior does not execute these deleted paths because it validates the availability of the required resource before analyzing the input conditions.
- Give the behavior representation the ability to hold parameters in the condition statements for solving problems like Sussman's anomaly [42] without breaking a plan into sub-plans. This also has the potential for increasing the efficiency of the monitoring mechanism.
- The Behavior Executive can be double tasked for predictive hierarchy generation that can report on a utility for accomplishing the requested plan. This is suggested in [23] as a required ability for multi-robot communication and coordination. Additionally, the use and benefits of a predictive element within a hybrid architecture is presented in [32].
  - May require statistics on past performance of generated hierarchies that accomplish the requested task.
  - May also require a utility function within the behavior representation for accomplishing its abstract goals
- When resources become unavailable, it triggers a replanning event. Defining more mechanism like anomaly and failure detection (suggested in Section 5.3) that trigger replanning events is ideal for creating a more adaptive system to unpredictable environments.

6.4 Final Remarks

Technological advances have made it possible for quicker processing and more concurrent threads of execution within autonomous mobile robot systems. Researchers capitalize on these advances to perform more deliberative tasks during concurrent re-

active execution. Therefore, more complex robot architectures are created that aim to reduce the human-in-the-loop control. This move to more complex architectures calls for the merging of deliberative planning with reactive control through a layered hybrid architecture design. The three layer architecture (TLA) design and implementation described throughout this document adheres to the integrity of the TLA paradigm through the functional decomposition of each layer based on abstraction level and computational complexity. Implemented within this architecture is the proposed behavior representation and dynamic behavior hierarchy generation algorithm. These components are shown to allow for a modular and robust implementation by defining an abstract link betwen the Sequencer and the Controller that seamlessly couples the two layers with loose dependence. The behavior representation is presented as a semantic suggestion rather then a syntactic implementation burden leaving the low-level details to the behavior programmer. The semantic suggestion is of how the behavior functions but does not restrict or dictate its implementation. Additionally, the hierarchy generation algorithm suggests a general process for using the behavior representation in translating high-level goals to the arbitrated behavior hierarchies that, given enough time, accomplish the high-level task. The proposed representation, suggested hierarchy generation algorithm, and architecture implementation are structured for programmer interpretation and maximum upgradability. Robustness and modularity is achieved through the functional decomposition of architectural components and abstract interfaces that loosely couple the components together. This creates a system-of-systems implementation that requires minimal reprogramming for system modifications.

Languages that are created to merge planning and execution require specific syntax and implementation. The behavior representations described in this appendix represent an abstract description that provides a semantic suggestion of how the behavior functions, but does not restrict or dictate its implementation. Therefore, these behaviors are the representations of the implemented behaviors used for the experiment discussed in Chapter V.

go-to-xy	
Initial Conditions:	
Active:	
Passive:	
Post Conditions:	
Adders:	*target-x-location*
	target-y-location
	all-stop
Deleters:	
Required Data:	
Goals Achieved:	Go-To-XY
Action Settings:	Vx
	TURNRATE
Vote:	1

Behavior Representation A.1: Behavior Representation for Behavior go-to-xy: Travels to a desired location (x, y) ignoring the destination heading.

go-to-xyt	
Initial Conditions:	
Active:	
Passive:	
Post Conditions:	
Adders:	*target-x-location*
	target-y-location
	target-t-location
	all-stop
Deleters:	
Required Data:	
Goals Achieved:	Go-To-XYT
Action Settings:	V_x
	TURNRATE
Vote:	1

Behavior Representation A.2: Behavior Representation for Behavior go-to-xyt: Travels to a desired location (x, y, θ), where θ is the difference in angle from the startup heading.

grab-object	
Initial Conditions:	
Active:	*gripper-open*
	gripper-outer-beam-broken
	gripper-inner-beam-broken
	all-stop
Passive:	
Post Conditions:	
Adders:	*gripper-closed*
	has-object
Deleters:	*gripper-open*
	not-has-object
Required Data:	GRIPPER
Goals Achieved:	Grab-Object
Action Settings:	TRANSLATE-GRIPPER
Vote:	7

Behavior Representation A.3: Behavior Representation for Behavior grab-object: Closes the gripper when an object has broken the beams.

laser-approach-object	
Initial Conditions:	
Active:	*gripper-open*
	treshold-min
	track-object
Passive:	
Post Conditions:	
Adders:	*gripper-outer-beam-broken*
	gripper-inner-beam-broken
	all-stop
Deleters:	*avoid-obstacle-target*
	avoid-obstacle
Required Data:	LASER
	GRIPPER
Goals Achieved:	Approach-Object
Action Settings:	Vx
	TURNRATE
Vote:	6

Behavior Representation A.4: Behavior Representation for Behavior laser-approach-object: Slowly approaches an object using LASER, places that object within the gripper beams, and stops.

laser-around-obstacle	
Initial Conditions:	
Active:	
Passive:	*treshold-min*
Post Conditions:	
Adders:	*avoid-obstacle-target*
Deleters:	*treshold-min*
Required Data:	LASER
Goals Achieved:	Avoid-Obstacle-Target
Action Settings:	Vx
	TURNRATE
Vote:	5

Behavior Representation A.5: Behavior Representation for Behavior laser-around-obstacle: Avoids obstacles by using the LASER to determine the best path to avoid the obstacle and advance toward the target location.

release-object	
Initial Conditions:	
Active:	
Passive:	*gripper-closed*
Post Conditions:	
Adders:	*gripper-open*
	not-has-object
Deleters:	*gripper-closed*
	has-object
Required Data:	GRIPPER
Goals Achieved:	Release-Object
Action Settings:	TRANSLATE-GRIPPER
Vote:	1

Behavior Representation A.6: Behavior Representation for Behavior release-object: Opens the gripper if closed.

sonar-approach-object	
Initial Conditions:	
Active:	*gripper-open*
	treshold-min
	track-object
Passive:	
Post Conditions:	
Adders:	*gripper-outer-beam-broken*
	gripper-inner-beam-broken
	all-stop
Deleters:	*avoid-obstacle-target*
	avoid-obstacle
Required Data:	SONAR
	GRIPPER
Goals Achieved:	Approach-Object
Action Settings:	Vx
	TURNRATE
Vote:	6

Behavior Representation A.7: Behavior Representation for Behavior sonar-approach-object: Slowly approaches an object using SONAR, places that object within the gripper beams, and stops.

sonar-around-obstacle	
Initial Conditions:	
Active:	
Passive:	*treshold-min*
Post Conditions:	
Adders:	*avoid-obstacle-target*
Deleters:	*treshold-min*
Required Data:	SONAR
Goals Achieved:	Avoid-Obstacle-Target
Action Settings:	Vx
	TURNRATE
Vote:	5

Behavior Representation A.8: Behavior Representation for Behavior sonar-around-obstacle: Avoids obstacles by using SONAR to determine the best path to avoid the obstacle and advance toward the target location.

track-object	
Initial Conditions:	
Active:	
Passive:	*visual-track-object*
Post Conditions:	
Adders:	*threshold-min*
Deleters:	
Required Data:	BLOBFINDER
Goals Achieved:	Track-Object
Action Settings:	Vx
	TURNRATE
Vote:	3

Behavior Representation A.9: Behavior Representation for Behavior track-object: Physically tracks (i.e. travels toward) an observed object until it is within a minimum range and stop.

visual-track-object	
Initial Conditions:	
Active:	
Passive:	
Post Conditions:	
Adders:	*visual-track-object*
Deleters:	
Required Data:	PTZ-CAMERA
	BLOBFINDER
Goals Achieved:	Visual-Track-Object
Action Settings:	PTZ
Vote:	1

Behavior Representation A.10: Behavior Representation for Behavior visual-track-object: Visually tracks an object by panning the camera to keep the object in the viewing window.

wall-follow	
Initial Conditions:	
Active:	
Passive:	
Post Conditions:	
Adders:	*avoid-obstacle*
	wall-follow
Deleters:	*approach-object*
	threshold-min
Required Data:	SONAR
Goals Achieved:	Explore
Action Settings:	VX
	TURNRATE
Vote:	2

Behavior Representation A.11: Behavior Representation for Behavior wall-follow: Travels along walls and keeps obstacles to the robot's left.

wander	
Initial Conditions:	
Active:	
Passive:	
Post Conditions:	
Adders:	*random-movement*
Deleters:	
Required Data:	
Goals Achieved:	Random-Movement
Action Settings:	Vx
	TURNRATE
Vote:	1

Behavior Representation A.12: Behavior Representation for Behavior wander: Travels in random 'S' patterns without regard to obstacles.

Example behaviors are used within this thesis to create a domain that is easy to follow and understand. The behavior representations in this appendix do not represent behaviors that were implemented and tested, but a set of behavior representations that can be implemented and should have the same results as described in this document. These behaviors are basic behaviors that serve just the purpose of the example domain. The behaviors in Appendix A present more general behaviors that showcase the planning capabilities of the proposed system.

avoid-obstacle			
Activation Path 1		*Activation Path 2*	
Initial Conditions:		**Initial Conditions:**	
Active:		*Active:*	
Passive:	*obstacle*	*Passive:*	*obstacle*
			target-location-set
Post Conditions:		**Post Conditions:**	
Adders:	*avoid-obstacle*	*Adders:*	*avoid-obstacle-target*
Deleters:	*obstacle*	*Deleters:*	*obstacle*
Required Data:	LASER	**Required Data:**	LASER
			MAP
Goals Achieved:	Obstacle-Avoidance	**Goals Achieved:**	Obstacle-Avoidance-Target
Action Settings:	Vx	**Action Settings:**	Vx
	TURNRATE		TURNRATE
Vote:	2	**Vote:**	10

Behavior Representation B.1: Behavior Representation for Example Behavior avoid-obstacle: avoids obstacles when an obstacle is detected and uses the optimal path to a target location if a target location is established.

deliver-object	
Initial Conditions:	
Active:	*has-object*
Passive:	*target-location-set*
Post Conditions:	
Adders:	*at-target-location*
Deleters:	
Required Data:	ODOMETRY
Goals Achieved:	Go-To-Target
	Goal-2
Action Settings:	Vx
	TURNRATE
Vote:	1

Behavior Representation B.2: Behavior Representation for Example Behavior `deliver-object`: Travels to a *target-location* when it has an object

get-object	
Initial Conditions:	
Active:	*gripper-open*
	tracking-object
Passive:	
Post Conditions:	
Adders:	*has-object*
	gripper-closed
Deleters:	*gripper-open*
	tracking-object
Required Data:	GRIPPERS
	CAMERA
	LASER
Goals Achieved:	Obtain-Object
Action Settings:	TRANSLATE-GRIPPER
Vote:	1

Behavior Representation B.3: Behavior Representation for Example Behavior `get-object`: Approaches a target object and picks it up

greeting	
Initial Conditions:	
Active:	
Passive:	
Post Conditions:	
Adders:	
Deleters:	
Required Data:	CAMERA
Goals Achieved:	Greet
Action Settings:	AUDIO-OUTPUT
Vote:	1

Behavior Representation B.4: Behavior Representation for Example Behavior `greeting`: Audibly greets recognized employees.

release-object	
Initial Conditions:	
Active:	
Passive:	*gripper-closed*
Post Conditions:	
Adders:	*not-have-object*
	gripper-open
Deleters:	*have-object*
Required Data:	GRIPPERS
Goals Achieved:	Release-Object
Action Settings:	TRANSLATE-GRIPPER
Vote:	3

Behavior Representation B.5: Behavior Representation for Example Behavior `release-object`: Releases an object by opening the GRIPPERS if they are closed.

scan-for-trash	
Initial Conditions:	
Active:	
Passive:	*scanned-object*
Post Conditions:	
Adders:	*tracking-object*
Deleters:	
Required Data:	CAMERA
Goals Achieved:	Target-Object
Action Settings:	CAMERA-PTZ
Vote:	5

Behavior Representation B.6: Behavior Representation for Example Behavior scan-for-trash: Scans for, identifies, and tracks trash (i.e. *tracking-object*) when found.

wall-follow	
Initial Conditions:	
Active:	
Passive:	*obstacle*
Post Conditions:	
Adders:	*map-explored*
Deleters:	*obstacle*
Required Data:	LASER
	MAP
Goals Achieved:	Explore
Action Settings:	Vx
	TURNRATE
Vote:	1

Behavior Representation B.7: Behavior Representation for Example Behavior wall-follow: Travels along walls and ensures all areas have been traversed.

Bibliography

1. Alami, Rachid, Raja Chatila, S. Fleury, Malik Ghallab, and François Félix Ingrand. "An Architecture for Autonomy". *International Journal of Robotics Research*, 17(4):315–337, APR 1998. PT:J.

2. Argall, Brenna, Brett Browning, and Manuela Veloso. "Learning to Select State Machines using Expert Advice on an Autonomous Robot". *Robotics and Automation, 2007 IEEE International Conference on*, 2124–2129, 10-14 April 2007. ISSN 1050-4729.

3. Arkin, Ronald C. *Robotics and Remote Systems for Hazardous Environments*, chapter Survivable Robotic Systems: Reactive and Homeostatic Control, pp. 135–154. Prentice-Hall, 1993.

4. Arkin, Ronald C. *Behavior-Based Robotics*, pp. 141–165. MIT Press, May 1998. ISBN 0-262-01165-4.

5. Bonasso, R. Peter, James Firby, Erann Gat, Kortenkamp David, David P. Miller, and Marc G. Slack. "Experiences with an Architecture for Intelligent, Reactive Agents". *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2/3):pp. 237–256, apr 1997.

6. Bonasso, R. Peter, David Kortenkamp, David P. Miller, and Marc G. Slack. "Experiences with an Architecture for Intelligent, Reactive Agents". *Intelligent Agents II: Agent Theories, Architectures, and Languages*, pp. 187–202. Springer-Verlag, 1995.

7. Braitenberg, Valentino. *Vehicles, experiments in synthetic psychology*. MIT Press, 1984.

8. Brooks, Rodney A. *A Robust Layered Control System For a Mobile Robot*. Technical report, Cambridge, MA, USA, 1985.

9. Connell, Jonathan H. "A behavior-based arm controller". *Robotics and Automation, IEEE Transactions on*, 5(6):pp. 784–791, 1989.

10. Connell, Jonathan H. *A Colony Architecture for an Artificial Creature*. Technical Report AITR-1151, Cambridge, MA, USA, 1989.

11. Connell, Jonathan H. "SSS: a hybrid architecture applied to robot navigation". *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pp. 2719–2724 vol.3. 1992.

12. Despouys, Olivier and François Félix Ingrand. "Propice-Plan: Toward a Unified Framework for Planning and Execution". *ECP '99: Proceedings of the 5th European Conference on Planning*, 278–293. Springer-Verlag, London, UK, 2000. ISBN 3-540-67866-2.

13. Duffy, Jeffrey P. and Gilbert L. Peterson. "An Abstract Behavior Representation for Robust, Dynamic Sequencing in a Hybrid Architecture". *Proc. of Spring AAAI Symposium on Architectures for Intelligent Theory-Based Agents*, March 2008.

14. Fetzek, Charles A. *Behavior-Based Power Management in Autonomous Mobile Robots*. Master's thesis, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, March 2008. AFIT/GCE/ENG/08-05.

15. Fikes, Richard E. and Nils J. Nilsson. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Technical Report 43r, AI Center, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, May 1971. SRI Project 8259.

16. Firby, R. James. *Adaptive Execution in Complex Dynamic Worlds*. Technical Report YALEU/CSD/RR #672, Yale University, January 1989.

17. Gat, Erann. "On Three-Layer Architectures". *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pp. 195–210, 1998.

18. Gat, Erran. "ESL: A Language For Supporting Robust Plan Execution in Embedded Autonomous Agents". *Aerospace Conference, 1997. Proceedings., IEEE*, volume 1, pp. 319–324 vol.1. 1997.

19. Gerkey, Brian P., Richard T. Vaughan, and Andrew Howard. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems". *In Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pp. 317–323, July 2003.

20. Ghallab, Malik, Rachid Alami, Joachim Hertzberg, Maria Gini, Maria Fox, Brian Williams, Bernd Schattenberg, Daniel Borrajo, Patrick Doherty, Jos M. Morina, Araceli Sanchis, Patrick Fabiani, and Martha Pollack. "A Roadmap for Research in Robot Planning", 2003. URL `http://www9.in.tum.de/research/tcu/roadmap.pdf`.

21. Ghallab, Malik, Adele Howe, Craig A. Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wikins. *PDDL—The Planning Domain Definition Language*. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, October 1998.

22. Hayes-Roth, Barbara, Karl Pfleger, Philippe Morignot, and Philippe Lalanda. *Plans and Behaviors in Intelligent Agents*. Technical Report KSL-95-35, Knowledge Systems Laboratory, Stanford University, March 1995.

23. Hooper, Daylond J. *A Hybrid Multi-Robot Control Architecture*. Master's thesis, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, December 2007. AFIT/GCS/ENG/08-02.

24. Ingrand, François Félix, Raja Chatila, Rachid Alami, and Frédéric Robert. "PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots".

Proc. of the IEEE Int. Conf. on Robotics and Automation, pp. 43–49. Minneapolis, 1996.

25. Ingrand, François Félix, Michael P. Georgeff, and Anand S. Rao. "An architecture for Real-Time Reasoning and System Control". *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, 1992. ISSN 0885-9000.

26. Kaelbling, Leslie P. *An Architecture For Intelligent Reactive Systems*. Technical Report 400, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1986.

27. Konolige, Kurt. *Saphira Robot Control Architecture: Saphira Version 8.1.0*. Technical report, SRI International, Menlo Park, California, April 2002.

28. Konolige, Kurt and Karen Myers. "The Saphira Architecture For Autonomous Mobile Robots". *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pp. 211–242, 1998.

29. Konolige, Kurt, Karen Myers, Enrique Ruspini, and Alessandro Saffiotti. "The Saphira Architecture: A Design for Autonomy". *Journal of Experimental and Theoretical AI, Special Issue on Software Architectures for Autonomous Agents*, 9, 1997.

30. Lemai, Solange and François Félix Ingrand. "Interleaving Temporal Planning and Execution: IxTeT-eXeC". *Proceedings of ICAPS'03 Workshop on Plan Execution*. Trento, Italy, June 2003.

31. Maes, Patti. "Situated Agents Can Have Goals". *Designing Autonomous Agents*, volume 6, pp. 49–70. MIT Press, 1990.

32. McDermott, Drew. *A Reactive Plan Language*. Technical Report YALE/DCS/TR-864, 1991.

33. MobileRobots Inc. *ARIA Overview*, 2.5.1 edition, 2007. URL `http://robots.mobilerobots.com/ARIA/`.

34. Müller, Armin and Michael Beetz. "Designing and Implementing a Plan Library for a Simulated Household Robot". Michael Beetz, Kanna Rajan, Michael Thielscher, and Radu Bogdan Rusu (editors), *Cognitive Robotics: Papers from the AAAI Workshop*, Technical Report WS-06-03, 119–128. American Association for Artificial Intelligence, Menlo Park, California, 2006. ISBN 978-1-57735-285-3.

35. Nguyen, XuanLong and Subbarao Kambhampati. "Reviving Partial Order Planning". *Proc IJCAI-2001*, pp. 459–466. 2001.

36. Nourbakhsh, Illah. "Using Abstraction to Interleave Planning and Execution". *Proceedings, Third Biannual World Automation Congress (WAC '98)*. 1998.

37. Pfleger, Karl and Barbara Hayes-Roth. "Plans Should Abstractly Describe Intended Behavior". Alex Meystel, Jim Albus, and R. Quintero (editors), *Intelligent*

Systems: A Semiotic Perspective, Proceedings of the 1996 International Multidisciplinary Conference, volume I: Theoretical Semiotics, 29–34. NIST, Gaithersburg, Maryland, 1996.

38. Pfleger, Karl and Barbara Hayes-Roth. *Using Abstract Plans to Guide Behavior*. Technical Report KSL-98-02, Computer Science Department, Stanford University, 1997.

39. Ranganathan, Ananth and Sven Koenig. "A reactive robot architecture with planning on demand". *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, 2:1462–1468 vol.2, 27-31 Oct. 2003.

40. Rosenblatt, Julio K. "Utility Fusion: Map-Based Planning in a Behavior-Based System". *FSR'97 International Conference on Field and Service Robotics*. December 1997.

41. Rosenblatt, Julio K. *Field and Service Robotics*, chapter Utility Fusion: Map-Based Planning in a Behavior-Based System, pp. 411–418. Springer-Verlag, 1998.

42. Russell, Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

43. Scheutz, Matthias and Virgil Andronache. "Architectural mechanisms for dynamic changes of behavior selection strategies in behavior-based systems". *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, 34(6):2377–2395, Dec. 2004. ISSN 1083-4419.

44. Schoppers, M. J. "Universal Plans for Reactive Robots in Unpredictable Environments". John McDermott (editor), *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, 1039–1046. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, Milan, Italy, 1987.

45. Simmons, Reid. "Coordinating planning, perception, and action for mobile robots". *SIGART Bull.*, 2(4):156–159, 1991. ISSN 0163-5719.

46. Simmons, Reid, Richard Goodwin, Karen Zita Haigh, Koenig Sven, and Joseph O'Sullivan. "A Layered Architecture for Office Delivery Robots". *First International Conference on Autonomous Agents*, pp. 235 – 242. February 1997.

47. Simmons, Reid, Richard Goodwin, Karen Zita Haigh, Koenig Sven, Joseph O'Sullivan, and Manuela Veloso. "Xavier: Experience with a Layered Robot Architecture". *SIGART Bull.*, 8(1-4):pp. 22–33, 1997. ISSN 0163-5719.

48. Simmons, Ried and David Apfelbaum. "A Task Description Language for Robot Control". *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, 1931–1937 vol.3. 1998.

49. Tang, Hongru, Aiguo Song, and Xiaobing Zhang. "Hybrid Behavior Coordination Mechanism for Navigation of Reconnaissance Robot". *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, 1773–1778, Oct. 2006.

50. Thrun, Sebastian, M. Bennewitz, W Burgard, A.B. Cremers, Frank Dellaert, Dieter Fox, D. Haehnel, Chuck Rosenberg, Nicholas Roy, Jamieson Schulte, and D. Schulz. "MINERVA: A second generation mobile tour-guide robot". 1999.

51. Williams, Brian C., Michel D. Ingham, Seung H. Chung, and Paul H. Elliott. "Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers". *Proceedings of the IEEE*, 91(1):212–237, January 2003.

52. Woolley, Brian. *Unified Behavior Framework for Reactive Robot Control in Real-time Systems*. Master's thesis, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, March 2007. AFIT/GCS/ENG/07-11.

53. Woolley, Brian and Gilbert Peterson. "Genetic Evolution of Hierarchical Behavior Structure". *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pp. 1731–1738, July 2007.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2008	Master's Thesis	August 2006 — Mar 2008

4. TITLE AND SUBTITLE

Dynamic Behavior Sequencing
in a
Hybrid Robot Architecture

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Jeffrey P. Duffy, Captain, USAF

5d. PROJECT NUMBER

JON 07-107/07-138

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/08-03

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Jacob Campbell, Civ
AFRL/RNYN
2241 Avionics Circle
Wright-Patterson Air Force Base, OH 45433
(937) 255-6127 x4154; jacob.campbell@wpafb.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Hybrid robot control architectures separate plans, coordination, and actions into separate processing layers to provide deliberative and reactive functionality. This approach promotes more complex systems that perform well in goal-oriented and dynamic environments. In various architectures, the connections and contents of the functional layers are tightly coupled so system updates and changes require major changes throughout the system. This work proposes an abstract behavior representation, a dynamic behavior hierarchy generation algorithm, and an architecture design to reduce this major change incorporation process. The behavior representation provides an abstract interface for loose coupling of behavior planning and execution components. The hierarchy generation algorithm utilizes the interface allowing dynamic sequencing of behaviors based on behavior descriptions and system objectives without knowledge of the low-level implementation or the high-level goals the behaviors achieve. This is accomplished within the proposed architecture design, which is based on the Three Layer Architecture (TLA) paradigm. The design provides functional decomposition of system components with respect to levels of abstraction and temporal complexity. The layers and components within this architecture are independent of surrounding components and are coupled only by the linking mechanisms that the individual components and layers allow. The experiments in this thesis demonstrate that the: 1) behavior representation provides an interface for describing a behavior's functionality without restricting or dictating its actual implementation; 2) sequencing control algorithm utilizes the representation interface for accomplishing high-level tasks through dynamic behavior sequencing; 3) representation, control logic, and architecture design create a loose coupling, but defined link, between the planning and behavior execution layer of the hybrid architecture, which creates a system-of-systems implementation that requires minimal reprogramming for system modifications.

15. SUBJECT TERMS

artificial intelligence, control systems, robotics, automation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Gilbert Peterson, Ph.D. (ENG)
U	U	U	UU	141	19b. TELEPHONE NUMBER *(include area code)* (937) 255–6565, x4281; gilbert.peterson@afit.edu